DCE
Application Development Guide: Introduction and Style
DCE
Application Development Guide: Introduction and Style
First Edition (March 2001)

This edition, SC24-5907-00, applies to Version 1 Release 1 of z/OS DCE Base Services, z/OS DCE user Data Privacy (DES and CDMF), z/OS DCE User Data Privacy (CDMF) (program number 5694-A01), and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for reader's comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Information Development, Dept. G60
1701 North Street
Endicott, NY 13760-5553
United States of America

FAX (United States & Canada): 1+607+752-2327
FAX (Other Countries):
Your International Access Code +1+607+752-2327

IBMLink™ (United States customers only): GDLVME(PUBRCF)
Internet e-mail: pubrcf@vnet.ibm.com

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1994, 2001. All rights reserved.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
The following statements are provided by the Open Software Foundation.

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

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

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

Copyright © 1993, 1994 Open Software Foundation, Inc.

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

- © Copyright 1990, 1991 Digital Equipment Corporation
- © Copyright 1990, 1991 Hewlett-Packard Company
- © Copyright 1989, 1990, 1991 Transarc Corporation
- © Copyright 1990, 1991 Siemens Nixdorf Informationssysteme AG
- © Copyright 1990, 1991 International Business Machines Corporation
- © Copyright 1988, 1989 Massachusetts Institute of Technology
- © Copyright 1988, 1989 The Regents of the University of California

All Rights Reserved.

Printed in the U.S.A.

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

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

UNIX is a registered trademark of The Open Group in the United States and other countries.

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

DECstation 3100 and DECnet are trademarks of Digital Equipment Corporation.

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

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

AFS and Transarc are registered trademarks of the Transarc Corporation.

Episode is a trademark of the Transarc Corporation.

Ethernet is a registered trademark of Xerox Corporation.

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

MX300i is a trademark of Siemens Nixdorf Informationssysteme AG.

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

X/Open is a trademark of The Open Group in the U.K. and other countries.

PostScript is a trademark of Adobe Systems Incorporated.

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

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

NOTICE: Notwithstanding any other lease or license that may pertain to, or accompany the delivery of, this computer software, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Section 52.227-19 of the FARS Computer Software-Restricted Rights clause.

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

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

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

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

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

### About This Book
- About This Book  
- Who Should Use This Book  
  - DCE Application Development Environment  
  - Unsupported OSF DCE Functions  
- How This Book Is Organized  
- Terminology Used in This Book  
- Conventions Used in This Book  
- Where to Find More Information  
  - Softcopy Publications  
  - Internet Sources  
  - Using LookAt to Look up Message Explanations  
  - Accessing Licensed Books on the Web

### Chapter 1. Introduction to DCE Application Programming
- About DCE Programming Style  
  - Mechanism, Policy, and Style  
  - Policy and Style Issues  
  - General Policies  
- Application Development Overview  
- Overview of DCE Application Development Steps  
- Application Development Tools  
  - DCE UUID Generator  
  - DCE Interface Definition Language  
  - DCE IDL Compiler  
  - Attribute Configuration File  
  - DCE Host Daemon  
  - DCE Control Program  
  - DCE API  
- The Interface Definition  
  - Generate the Interface UUID  
  - Write the Interface Definition File  
  - Write the Attribute Configuration File (ACF)  
  - Process the Files with the IDL Compiler  
- The Server Initialization  
  - Set Up the Server's Objects  
  - Set Up Security  
  - Define Manager Entry Point Vectors for Each Set of Operations  
  - Register the Server  
  - Specify Multithreadedness  
  - Listen for Incoming Service Requests  
  - Clean Up Code When the Server Terminates  
- The Client Binding and RPC Invocation  
  - Import the Binding Information from the Namespace  
  - Annotate the Binding Handle for Security  
  - Invoke Remote Procedure Calls  
- The Server's Manager of RPC Requests  
  - Get the Client's Credentials  
  - Get RACF Authorization using RACF-DCE Interoperability  
  - Get the Object's Access Control List (ACL)  
  - Make the Authorization Decision

© Copyright IBM Corp. 1994, 2001
Service the RPC Request ............................................  28
Return the Results and Resume Listening ..................................  28
Writing a Simple Distributed Application on z/OS ..............................  29
  High-Level Application Development Steps ..................................  29
  Step 1. Creating Files for Your Application ..................................  30
  Step 2. Generating a UUID and IDL File ...................................  34
  Step 3. Naming the Interface for Your Application. .............................  37
  Step 4. Defining the Interface Operations ...................................  37
  Step 5. Compiling the Interface with the IDL Compiler ...........................  37
  Step 6. Writing Your Server and Manager Code ..................................  39
  Step 7. Writing the Client Code for Your Application ..................................  40
  Step 8. Compiling the Client and Server Programs ..................................  40
  Step 9. Link-Editing Your Application .....................................  45
  Step 10. Building Your DCE Application ...................................  48
Running Your DCE Application ........................................  50
  Step 1. Starting Your Server ..........................................  50
  Step 2. Starting Your Client ...........................................  51
  Step 3. Checking Your Application .......................................  52
  Step 4. Stopping Your Application .......................................  52
  Step 5. Setting Environment Variables ....................................  53
Language Environment Runtime Options Considerations ..........................  57
Note on Interlanguage Calls ..........................................  58
Creating a Sample Application: GREET .....................................  58
  Fast Path to Running Greet ...........................................  58
  1. Creating Files for the Greet Application ..................................  59
  2. Generating a UUID and an IDL File ....................................  59
  3. Naming the Greet Interface .........................................  60
  4. Defining the Interface Operations ......................................  61
  5. Compiling the Greet Interface with the IDL Compiler ...........................  61
  6. Writing the Greet Server and Manager Code ..................................  63
  7. Writing the Greet Client Code ........................................  66
  8. Building the Greet Client and Server Programs ..................................  69
  9. Starting the Greet Server ..........................................  72
  10. Starting the Greet Client ........................................  73

Chapter 2. Extending the Greet Application ....................................  75
Logging Into DCE ..................................................  75
  Changing Your Login Context .........................................  76
  Inheriting Contexts for Multiple Programs ..................................  78
Searching for Your Server .............................................  78
  Using the DCE Host Daemon ...........................................  79
  Using the Name Service Interface .......................................  86
  Importing the Greet2 Server Binding from the Namespace .........................  91
  Starting the Greet2 Server and Client .....................................  94
  Monitoring Your Distributed Application ....................................  96
Using Automatic Binding Handles .........................................  97
  Modifying the Greet Code ............................................  98
Adding Multiple Object Types .......................................... 100
  Modifying the Greet IDL File ......................................... 102
  Modifying the Greet4 Manager ........................................ 103
  Modifying the Greet4 Server ........................................ 104
  Modifying the Greet4 Client .......................................... 110
  Starting the Greet4 Server and Client .................................... 114
Chapter 3. Securing the Greet Application .................................................. 117
Greet with Name-Based Authorization .............................................................. 117
  Finding the Source Code for this Example ................................................... 117
Greet Server (Main) ............................................................................................ 118
Greet Server (Manager) ....................................................................................... 123
Greet Client ........................................................................................................ 125
  Creating a Key Table File for the Greet5 Server .............................................. 128
  Starting the Greet5 Application ...................................................................... 129
Greet5 Server Output .......................................................................................... 130
Greet5 Client Output .......................................................................................... 131
Greet with EPAC-based Authorization ............................................................... 131
  Finding the Source Code for this Example ..................................................... 134
  Building the Greet6 Example ...................................................................... 136
  Starting the Greet6 Server .......................................................................... 141
  Updating the ACL Database File .................................................................. 142
  Starting the Greet6 Client .......................................................................... 144

Chapter 4. Threads ............................................................................................. 149
Thread Use Policy ............................................................................................... 150
  To Thread or Not to Thread ........................................................................ 150
  How Many Threads? .................................................................................... 151
  Scheduling Policies ...................................................................................... 151
Thread Safety ...................................................................................................... 152
  Thread Rules ................................................................................................. 152
Threads Programming Topics ............................................................................ 153
  Thread Handles ............................................................................................ 154
  Storage for Thread Specific Data ................................................................. 154
  Canceling Threads ........................................................................................ 155
  Signals ........................................................................................................... 159
  Forking in a Threaded Application ............................................................... 161
RPC Threads ........................................................................................................ 162
  RPC Cancel Semantics .............................................................................. 162

Chapter 5. Security ............................................................................................. 163
  The Basic Security Model ............................................................................ 163
  Application Roles .......................................................................................... 165
Authentication Model .......................................................................................... 165
  The DCE Authentication Model .................................................................. 165
  Application-Level Authentication ............................................................... 166
  Obtaining an Authentication Identity ........................................................... 168
  The Authenticated RPC Call ................................................................... 169
  Managing Keys ............................................................................................. 169
  Default Server Authentication Steps ........................................................... 170
  Default Client Authentication Steps ........................................................... 174
Authorization ......................................................................................................... 176
  Authorization using RACF-DCE Interoperability ....................................... 177
  Client Credentials ....................................................................................... 179
  Access Control Lists .................................................................................... 180
  ACL Managers ............................................................................................. 182

Chapter 6. Binding ............................................................................................. 203
  The Binding Model ........................................................................................ 203
  Server Binding Model ................................................................................... 204
  Client Binding Model .................................................................................... 205
Figures

1. The Combined Effect of IDL and the RPC Runtime ........................... 5
2. IDLALLOC Utility Menu .................................................................. 34
3. Example of an IDL Template .......................................................... 34
4. Sample Shell Command to Run the UUID Generator ......................... 35
5. Sample Shell Command to Create a UUID for STOCK Application .......... 35
6. Sample Shell Command to Create UUID Using a PDS .......................... 35
7. Sample TSO/E Command to Create a UUID for STOCK Application ........ 35
8. Sample TSO/E Command to Create a UUID using an HFS File ............... 36
9. Sample JCL to Run the UUID Generator .......................................... 36
10. Sample JCL to Create a UUID for STOCK Application ....................... 36
11. Sample JCL to Create UUID Using an HFS file ................................ 36
12. Sample Shell Command to Run the IDL Compiler ........................... 38
13. Sample Shell Command to Run the IDL Compiler — HFS File ................. 38
14. Sample Shell Command to Run the IDL Compiler — PDS File ............... 38
15. Sample TSO/E Command to Run the IDL Compiler .......................... 39
16. Sample JCL to Run the IDL Compiler in Batch ................................ 39
17. Compiling Your Client using HFS Files .......................................... 41
18. Compiling Your Server using HFS Files ......................................... 41
19. Compiling Your Client using PDS Files .......................................... 42
20. Compiling Your Server using PDS Files .......................................... 42
21. Compiling in the TSO/E environment .............................................. 42
22. Sample JCL to Compile Your DCE Application ................................ 44
23. Sample Shell Command to Link Your Client using HFS Files ................. 45
24. Sample Shell Command to Link Your Server using HFS Files ................. 46
25. Sample Shell Command to Link Your Client using PDS ....................... 46
26. Sample Shell Command to Link Your Server using PDS ....................... 46
27. Sample TSO/E Command to Link Your Client .................................. 48
28. Sample JCL to Link Your DCE Client ............................................ 47
29. Sample JCL to Link Your DCE Server ............................................ 48
30. Sample Makefile to Build Your DCE Application .................................. 49
31. Example Command for Running Your DCE Server in Background Mode ...... 50
32. Example JCL for Running Your DCE Server ..................................... 51
33. Example Command for Running Your DCE Client ............................ 51
34. Example JCL for Running Your Client ............................................. 51
35. Example Environment Variables ...................................................... 56
36. Specifying Environment Variables in Batch ..................................... 56
37. Shell Command to run the UUID Generator for Greet ........................ 59
38. Generating a UUID for the Greet Application in Batch ....................... 60
39. Example of the Greet Application IDL File ..................................... 60
40. Naming the Greet Interface ......................................................... 60
41. The Greet Application Interface Operations .................................... 61
42. Sample Command to Invoke the IDL Compiler in the Shell ................... 62
43. Sample TSO/E Command to Run the IDL Compiler ........................... 62
44. Sample JCL to Run the IDL Compiler ............................................. 62
45. The Greet Application Header File ............................................... 63
46. Greet Manager Source Code ....................................................... 64
47. Greet Server Source Code ......................................................... 65
48. Greet Client Source Code ........................................................... 68
49. Makefile to Build the Greet Application .......................................... 69
50. Sample JCL to Compile Greet ......................................................... 70
### Tables

1. User-Written Files for DCE Applications ........................................... 30
2. System-generated Files for DCE Applications ..................................... 31
3. Recommended DCE Application Data Set Attributes ............................. 33
4. IDL Compiler Output ........................................................................... 37
5. Cancelability State ............................................................................. 156
6. Authentication ..................................................................................... 164
7. Binding Semantics (Client Operation) ..................................................... 213
8. Examples of Objects (Server Resources) ............................................... 227
9. RPC API Parameter Semantics ............................................................... 242
10. IDL/NDR/C Type Mappings ................................................................. 245
11. Timop Server Source Files ................................................................. 275
12. Timop Client Source Files ................................................................. 275
13. The Timop Source Files (HFS and PDS Names) .................................... 277
About This Book

The objective of this book is to assist you in designing, writing, compiling, linking, and running distributed applications on the IBM z/OS operating system. Specifically, use this book for creating applications with z/OS DCE running on the stand-alone z/OS system. The steps to develop a distributed application using DCE services and application programming interfaces (API) are described in progressive detail. Also discussed are the development decisions and tools that you need to consider when developing your distributed application using z/OS DCE.

To create DCE applications that access IMS™ or CICS® transactions, refer to z/OS DCE Application Support Programming Guide.

Who Should Use This Book

This book assumes you are an experienced application developer or programmer with a working knowledge of the C programming language and the z/OS operating system. You do not have to possess prior knowledge of, or experience with, designing and writing distributed applications using the Open Software Foundation (OSF) Distributed Computing Environment (DCE) services and APIs.

Ideally, you should be able to:

- Allocate z/OS data sets
- Edit, browse, and copy z/OS data sets and associated members
- Print data sets
- Write and submit batch jobs on z/OS
- Write, compile, link, and run C/C++ programs on z/OS
- Write and understand JCL to run on z/OS
- Understand Shell and TSO/E commands.

A good working knowledge and understanding of the following would be helpful:

- Interactive System Productivity Facility/Program Development Facility (ISPF/PDF)
- Concepts behind a distributed application
- Using the Spool Display and Search Facility (SDSF) to check on the status of your application.

Some exposure to the UNIX or AIX® operating system is helpful but not essential to use this book.

You should be familiar with the concepts of the Distributed Computing Environment. If you are not, read z/OS DCE Introduction.

DCE Application Development Environment

It is conceivable that you may develop your DCE applications on a platform other than the z/OS operating system. Perhaps you may prefer to work on a UNIX-based workstation or a proprietary operating system. If your goal is to ultimately run either the client or server portion of your DCE application on z/OS, ensure that portion of your DCE application conforms to all recommendations contained in this book.

This book describes the development steps assuming you are developing your DCE applications on the z/OS operating system. If you are developing DCE applications on the z/OS platform that are targeted to run on another platform, consult the DCE application development documentation associated with that platform.
Unsupported OSF DCE Functions

The following DCE technology functions, which may be available in the Distributed Computing Environment product from OSF or on DCE offerings from other vendors, are not supported in z/OS DCE:

- DCE Directory Services
  - X/Open Data Services (XDS) function (Global Directory Service (GDS) portion)
  - X/Open OSI-Abstract-Data Manipulation (XOM) function (GDS portion)
  - Global Directory

On z/OS, only CDS, XDS, and XOM access to CDS are supported. GDS, XDS, and XOM access to GDS are not supported.

The following DCE daemon is not supported on z/OS DCE:

- DCE Security daemon

  Note: Although the DCE Security daemon is not included in z/OS DCE, a security server is available from IBM as a separately licensed product.

OSF DCE Programming Interfaces: The following programming interfaces are not supported:

- pthread interfaces
  - The following interfaces are not supported by z/OS DCE and return -1, errno ENOSYS:
    - pthread_attr_getinheritsched()
    - pthread_attr_getprio()
    - pthread_attr_getsched()
    - pthread_attr_setinheritsched()
    - pthread_attr_setprio()
    - pthread_attr_setsched()
    - pthread_getprio()
    - pthread_getscheduler()
    - pthread_setprio()
    - pthread_setscheduler()
  - For all pthread interfaces (including mutexes, threads, condition variables and so on), the interfaces do not accept copies of the objects as a parameter. The object returned from the pthread interface to create the object must be used at all times.
  - Unlike the OSF DCE implementation, the z/OS DCE implementation of the following functions can raise an exception (exc_e_cpa_error) in error situations:
    - pthread_lock_global_np()
    - pthread_unlock_global_np()
  - pthread_cond_timedwait() expects an absolute hardware time (that is, time-of-day clock value) for the wait time instead of the DCE software clock time, which is what OSF/DCE expects. pthread_get_expiration_np() returns a software adjusted time as in the OSF/DCE model, and is used as input to pthread_cond_timedwait().
  - exc_report() does not print out a message to stderr as expected. z/OS DCE uses Reliability, Availability and Serviceability (RAS) services to log messages instead of this function.

- Exceptions
  - z/OS DCE catches z/OS abends in addition to the set of predefined exceptions and user defined exceptions.
TRY/CATCH/ENDTRY macros can raise an exc_e_insfmem exception if they cannot get enough heap storage.
TRY/CATCH/ENDTRY macros can raise an exc_e_uninitexc exception if they detect that the CATCH does not specify a valid exception.

Remote Procedure Call
- rpc_cs_binding_set_tags()
- rpc_cs_char_set_compat_check()
- rpc_cs_eval_with_universal()
- rpc_cs_eval_without_universal()
- rpc_cs_get_tags()
- rpc_mgmt_set_server_stack_size()
- rpc_ns_import_ctx_add_eval()
- rpc_ns_mgmt_free_attr_data()
- rpc_ns_mgmt_free_codesets()
- rpc_ns_mgmt_read_attr_begin()
- rpc_ns_mgmt_read_attr_done()
- rpc_ns_mgmt_read_attr_next()
- rpc_ns_mgmt_remove_attribute()
- rpc_ns_mgmt_read_codesets()
- rpc_mgmt_set_attribute()
- rpc_rgy_get_codesets()
- rpc_rgy_get_max_bytes()

Security Services
- sec_login_get_pwent()
- sec_login_init_first()

How This Book Is Organized

This guide is divided into nine chapters and three appendixes containing program samples. The first part, designed for novice DCE application programmers, introduces DCE application programming with z/OS DCE. The following parts contain detailed information on using the various DCE components and their respective APIs that are supported by z/OS DCE.

Chapter 1, “Introduction to DCE Application Programming” on page 1 introduces DCE and its various components as they relate to application programming. It shows you the steps to create a distributed application using z/OS DCE with several example applications as templates for development. This puts you on a fast path to developing DCE applications on the z/OS operating system.

Chapter 2, “Extending the Greet Application” on page 75 and Chapter 3, “Securing the Greet Application” on page 117 expand on the example application showing you how to exploit DCE services such as security and automatic binding.

Chapter 4, “Threads” on page 149 introduces the use of DCE Threads in your distributed applications. Discussed are how and when to use threads, including thread rules, safety, and related programming topics such as storage use with threads. The interaction of RPC with threads is also explained.

Chapter 5, “Security” on page 163 introduces the basic DCE security model, including the two main security services: authentication and authorization. The roles of application programs are emphasized, with examples of routines used by both clients and servers to set up and implement security functions such as acquiring and validating credentials and performing password management.

Chapter 6, “Binding” on page 203 describes the process by which clients establish relationships with servers in DCE applications. The binding model, call routing, binding handles, methods and management are covered.
• Chapter 7, “Using the DCE Name Service” on page 215 introduces the DCE Name Service. The naming, organization and retrieval of object and binding information is covered, including the use of the DCE Cell Directory Service (CDS) as the primary name database. The interaction of the name service with various binding methods and the necessary routines used by application programs are outlined.

• Chapter 8, “RPC Parameters” on page 241 explains the DCE RPC mechanism and how its data model differs from that of a local call model, particularly focusing on RPC parameter syntax. It contains style and policy recommendations for data passing, with numerous examples.

• Chapter 9, “Server Management” on page 267 briefly outlines basic DCE server management operations and how they are supported by an application.

• Appendix A, “A Sample Application” on page 271 is a discussion of an example DCE application, with both server and client portions, that illustrates the development steps and programming policies in the foregoing chapters.

• Appendix B, “Another Sample DCE Application: TIMOP” on page 273 is another example of a simple but complete DCE application program.

• Appendix C, “Greet6 ACL Manager Example” on page 303 is an example of using a simple ACL manager, one of the basic security features discussed in Chapter 5, “Security” on page 163.

To find more information on topics related to application development not addressed in this book, consult the following:

- z/OS DCE Application Development Reference SC24-5908
- z/OS DCE Administration Guide SC24-5904
- z/OS DCE Command Reference SC24-5909
- z/OS DCE Application Support Programming Guide SC24-5902 (CICS and IMS)
- z/OS DCE Messages and Codes SC24-5912

For example, the DCE CDS is discussed in detail as a separate component in the administration documentation. Similarly, certain aspects of the DCE Security Service important to application developers (such as adding new principals to the registry database) are found only in the administration books.

### Terminology Used in This Book

Because DCE technology has been developed from the UNIX environment, many DCE concepts and terms contained herein relate to that environment. z/OS terms and concepts are used throughout this book wherever possible.

The following table explains how certain terms are used in this book and how they are related.

<table>
<thead>
<tr>
<th>Related Terms</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Throughout this book, the term file can refer to a sequential data set, a</td>
</tr>
<tr>
<td>data set</td>
<td>member of a partitioned data set, or a hierarchical file system (HFS) file.</td>
</tr>
<tr>
<td>sequential data set</td>
<td>For more information on hierarchical file systems in z/OS, see z/OS UNIX</td>
</tr>
<tr>
<td>hierarchical file system (HFS) file</td>
<td></td>
</tr>
</tbody>
</table>
Conventions Used in This Book

This book uses the following typographic conventions:

**Bold**  
**Bold** words or characters represent system elements that you must enter into the system literally, such as commands, options, or path names.

*Italic*  
*Italic* words or characters represent values for variables.

**Example font**  
Examples and information displayed by the system appear in constant width type style.
Brackets enclose optional items in format and syntax descriptions.

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

A vertical bar separates items in a list of choices.

Angle brackets enclose the name of a key on the keyboard.

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

A backslash is used as a continuation character when entering commands from the shell that exceed one line (255 characters). If the command exceeds one line, use the backslash character \ as the last non-blank character on the line to be continued, and continue the command on the next line.

This book uses the following keying conventions:

\<Alt-c> The notation <Alt-c> followed by the name of a key indicates a control character sequence.

\<Return> The notation <Return> refers to the key on your keyboard that is labeled with the word Return or Enter, or with a left arrow.

**Entering commands** When instructed to enter a command, type the command name and then press <Return>.

---

**Where to Find More Information**

Where necessary, this book references information in other books using shortened versions of the book title. For complete titles and order numbers of the books for all products that are part of z/OS, see the [Z/OS Information Roadmap](http://www.ibm.com/servers/eserver/zseries/zos/bkserv/SA22-7500). For complete titles and order numbers of the books for z/OS DCE, refer to the publications listed in the "Bibliography" on page 353.

For information about installing z/OS DCE components, see the [z/OS Program Directory](http://www.ibm.com/servers/eserver/zseries/zos/bkserv/).

**Softcopy Publications**

The z/OS DCE library is available on a CD-ROM, *z/OS Collection*, SK3T-4269. The CD-ROM online library collection is a set of unlicensed books for z/OS and related products that includes the IBM Library Reader™. This is a program that enables you to view the BookManager® files. This CD-ROM also contains the Portable Document Format (PDF) files. You can view or print these files with the Adobe Acrobat reader.

**Internet Sources**

The Softcopy z/OS publications are also available for web-browsing and for viewing or printing PDFs using the following URL:


You can also provide comments about this book and any other z/OS documentation by visiting that URL. Your feedback is important in helping to provide the most accurate and high-quality information.
Using LookAt to Look up Message Explanations

LookAt is an online facility that allows you to look up explanations for z/OS messages. You can also use LookAt to look up explanations of system abends.

Using LookAt to find information is faster than a conventional search because LookAt goes directly to the explanation.

LookAt can be accessed from the Internet or from a TSO command line.

You can use LookAt on the Internet at:

To use LookAt as a TSO command, LookAt must be installed on your host system. You can obtain the LookAt code for TSO from the LookAt Web site by clicking on the News and Help link or from the z/OS Collection, SK3T-4269.

To find a message explanation from a TSO command line, simply enter: lookat message-id as in the following:
lookat iec192i

This results in direct access to the message explanation for message IEC192I.

To find a message explanation from the LookAt Web site, simply enter the message ID and select the release with which you are working.

Note: Some messages have information in more than one book. For example, IEC192I has routing and descriptor codes listed in z/OS MVS Routing and Descriptor Codes, SA22-7624. For such messages, LookAt prompts you to choose which book to open.

Accessing Licensed Books on the Web

z/OS licensed documentation in PDF format is available on the Internet at the IBM Resource Link site:
http://www.ibm.com/servers/resourcelink

Licensed books are available only to customers with a z/OS license. Access to these books requires an IBM Resource Link user ID, password, and z/OS licensed book key code. The z/OS order that you received provides a memo that includes your key code.

To obtain your IBM Resource Link user ID and password, logon to:
http://www.ibm.com/servers/resourcelink

To register for access to the z/OS licensed books:
1. Logon to Resource Link using your Resource Link user ID and password.
2. Select User Profiles located on the left-hand navigation bar.
5. Supply your key code where requested and select the Submit button.

If you supplied the correct key code you will receive confirmation that your request is being processed.

After your request is processed you will receive an e-mail confirmation.
Note: You cannot access the z/OS licensed books unless you have registered for access to them and received an e-mail confirmation informing you that your request has been processed.

To access the licensed books:

1. Logon to Resource Link using your Resource Link user ID and password.
2. Select Library.
3. Select zSeries.
4. Select Software.
5. Select z/OS.
6. Access the licensed book by selecting the appropriate element.
Chapter 1. Introduction to DCE Application Programming

The first part of this chapter addresses some programming style issues of particular importance to DCE application developers, while the rest introduces the basic steps involved in writing a DCE application.

About DCE Programming Style

The Style Guide part of this book attempts to bridge a gap. On one side stands the reference material provided by the rest of the DCE Application Development Guide material and by the z/OS DCE Application Development Reference. In theory, this material provides complete documentation of the mechanisms of DCE application programming. In particular, it documents the syntax and semantics of every DCE API interface and IDL construct and provides a service-by-service guide to their use.

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

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

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

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

Mechanism, Policy, and Style

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

In practice, the distinction between mechanism and policy/style is often vague. The other parts of the z/OS DCE application development documentation set contain much that could be considered policy and style guidance. And, for reasons discussed in some detail in the next section, the Style Guide often contains descriptions of the mechanisms of DCE programming.
Nevertheless, the *Style Guide* does attempt to keep to the ground of policy and style issues. It assumes that you already know what mechanisms are available and attempts to provide guidance about the choices you have in using those mechanisms. One result is that the *Style Guide* is not a tutorial; it often assumes knowledge of terms and concepts that are explained elsewhere in the programmer's documentation. On the other hand, the *Style Guide* does in many cases provide high-level discussions of the organization and principals of DCE services, such as the security services. The assumption is that you may already know many of the details but may lack an overall framework. Often, such a general model is just what you need to be able to make rational policy decisions.

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

**Policy and Style Issues**

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

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

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

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

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

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

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

General Policies

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

- Servers are generalized providers of the services specified by their published (IDL) interfaces. That is,
servers should encapsulate the services they provide in such a way that naive clients, with no
knowledge of the specifics of server implementation, can successfully make use of these services via
the remote interfaces. In this sense, servers are much like libraries. One should not assume that
clients will be written by someone with knowledge of server internals. Where appropriate, define
wrapper routines for the IDL operations to shield developers from binding handles and other RPC
peculiarities.

- Servers should make their resources known to clients using standard mechanisms. In particular, they
should export their bindings according to the recommended service models, use name and endpoint
services rather than fixed bindings and well-known endpoints, and associate exported objects with
UUIDs.

- Clients and servers should be portable, using DCE provided mechanisms instead of operating system
and transport-dependent mechanisms. For example, data streams should be communicated via the
RPC pipe mechanism rather than socket calls. The AES/DC is the definitive guide to application
portability using the DCE mechanisms.

- Distributed applications make greater administrative demands than non-distributed ones. Clients and
servers need to be written with an eye to minimizing and simplifying administrative tasks. This means,
for example, that
  - Applications need to be as configuration and location independent as possible. In particular, this
  means giving careful thought to the use of name services for advertising and finding resources.
  - Applications require both local and DCE identities and privileges. They should follow the
    recommended models for acquiring and maintaining these privileges and identities.
  - Servers should be administratively interoperable; that is, they should behave like the standard
    DCE servers, exporting the recommended management interfaces, exporting ACL managers,
    logging errors and messages, and providing for the standard startup and shutdown mechanisms.

- Distributed security is inherently more complex than local system security (you cannot just “lock the
door”). Applications should follow the recommended security policies rigorously.
Clients and servers should follow the recommended internationalization guidelines to ensure character set interoperability.

**Application Development Overview**

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

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

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

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

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

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

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

The IDL compiler also generates a header file for inclusion in the server and client source files. It contains all the declarations that result from the IDL file definitions. Among these are, for example, the interface specification identifier, which will be used at runtime to describe the interface being defined in the programs.
Once you have linked the stub files (and the DCE library) to their respective client and server modules, the IDL-generated stubs make the client and server seem to communicate directly through the operation signatures you defined in the original .idl file, although in actuality client/server communications pass back and forth through layers of stub and runtime processing, which are necessary to send and receive the data over the network. Figure 1 illustrates how the combination of IDL (by means of the stubs it generates) and the RPC runtime routines shields both client and server from the details of network communications.

---

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

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

### Overview of DCE Application Development Steps

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

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

A. **CLIENT and SERVER:** Define the IDL interface [Steps A1 to A4]
B. **SERVER:** Set up and listen [Steps B2 to B8]
C. **CLIENT:** Bind to and invoke the server [Steps C1 to C4]
D. **SERVER:** Service request(s) [Steps D1 to D5]
Following is an overview list of all 21 steps, separated into the four main phases previously described. Each step's numeral is followed by a / (slash) and the terms Client and/or Server to indicate whether it applies to the application's server or client, or both.

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

B. **SERVER:** Initialization.
   - **B2/Server:** Set Up the Server's Objects.
   - **B3/Server:** Set Up Security.
   - **B4/Server:** Define the manager Entry Point Vectors (EPVs).
   - **B5/Server:** Register the Server.
   - **B6/Server:** Specify multithreadedness.
   - **B7/Server:** Listen for incoming service requests.
   - **B8/Server:** Cleanup When Server Terminates.

C. **CLIENT:** Bind to and invoke the server.
   - **C1/Client:** Multithreaded Client Design.
   - **C2/Client:** Import the binding information from the namespace (CDS).
   - **C3/Client:** Annotate the binding handle for security.
   - **C4/Client:** Invoke an RPC interface operation.

D. **SERVER:** Service the request.
   - **D1/Server:** Get the client's credentials.
   - **D2/Server:** Get the object's Access Control List (ACL).
   - **D3/Server:** Make the authorization decision.
Application Development Tools

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

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

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

- **Attribute Configuration Language**
  
  To allow developers to control the interface between local application code and the RPC interface, there is an optional attribute configuration language supported by the IDL compiler.

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

In addition, DCE provides an extensive application programming interface that includes routines for configuration, using threads, managing servers, and interacting with the other DCE services.

**DCE UUID Generator**

The UUID generator **uuidgen** is an interactive utility that creates UUIDs (Universal Unique Identifiers). A UUID is a hexadecimal number that contains information that makes it unique from all other UUIDs. Applications use UUIDs to identify many kinds of entities, including interface definitions. Consequently, application developers typically use the UUID generator when they are creating their interface definition files.

To run the UUID generator, issue the **uuidgen** command. This command offers several options, including an option to create a template interface definition file (an .idl file) containing a newly generated interface UUID. For complete information about generating UUIDs and template interface definition files, see the **z/OS DCE Application Development Guide: Core Components**. Refer to the **z/OS DCE Command Reference** for a description of the uuidgen utility and its options.
DCE Interface Definition Language

As was mentioned earlier in this chapter, developing a DCE application involves writing and compiling an interface definition, which defines the application's client/server interface. Application developers use the DCE Interface Definition Language (IDL) to write the interface definition. IDL is a high-level descriptive language whose syntax resembles that of ANSI C. IDL is a declarative, not a procedural, language.

Some of the important attributes specified with IDL are the following:

- For interfaces:
  - **uuid** Specifies the interface’s UUID.
  - **version** Specifies the interface major and minor version number.
- For parameters:
  - **in** Signifies a parameter whose value is passed from the client to the server.
  - **out** Signifies a parameter whose value is passed from the server to the client.
- For data types:
  - **handle** Specifies a customized binding handle. See Chapter 6, “Binding” on page 203 or a discussion of binding handles and binding methods in more detail.
  - **context_handle** Specifies a context handle, which is a pointer to state information that the server uses and which is maintained across RPC invocations. An example of a context handle is a file pointer. The z/OS DCE Application Development Guide: Core Components discusses context handles in more detail.

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

DCE IDL Compiler

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

**Note:** Stub files used on the z/OS DCE platform **must** be generated with the z/OS DCE IDL compiler.

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

You run the IDL compiler by issuing the `idl` command. See the `idl` information in the z/OS DCE Command Reference for a description of the `idl` command and its options.
Attribute Configuration File

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

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

For complete information on the set of ACF attributes, see the z/OS DCE Application Development Guide: Core Components.

DCE Host Daemon

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

The most important feature of the dced is that it provides the endpoint mapper service. This service maintains the host's local endpoint map for local RPC servers and looks up endpoints for RPC clients. An endpoint is the address of a specific instance of a server that is executing in a particular address space on a given host. Each endpoint can be used on a host by only one server at a time. The endpoint map is the system-specific database on each host, in which servers register their endpoints and associated addressing information (information about communication protocols, objects, and so on). A server registers separate endpoints for each of its RPC interfaces and any objects the server offers with the interface.

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

Other remote services of dced include host data management, server control, security validation, and key table management. These are described in detail in the z/OS DCE Application Development Guide: Core Components.
DCE Control Program

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

- For exporting binding information to a namespace, instead of putting C code in your application to call the NSI routines, rpc_ns_...(), you could write a dcecp script that calls rpcentry export and its related commands.

- For installation, you might need to create a principal name and/or set an ACL on it. Instead of writing C code in your application's initialization section to call sec_rgy_pgo_...() and sec_acl_...(), you could ship a dcecp script that does the following:

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

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

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

The dcecp can also be useful for debugging, as these examples show:

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

See the [z/OS DCE Command Reference](#) for more information about dcecp commands.

DCE API

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

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

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

To create an interface definition, follow these steps:

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

The next sections describe these steps in more detail.

Generate the Interface UUID

Interfaces, like most other objects and entities in DCE, are identified by associating each one with a 128-bit Universal Unique Identifier (UUID). An interface’s UUID serves to identify it throughout DCE. Every interface in a DCE application must have a UUID assigned to it.

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

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

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

or

```
uuidgen -i -o your_interface_name.idl
```

where `your_interface_name` is the name you have given your interface, and `.idl` is the suffix that all interface definitions use by convention. The `uuidgen` utility generates a file named `your_interface_name.idl`, that contains a skeleton of an interface definition and includes the newly generated UUID for the interface. See the [z/OS DCE Application Development Guide: Core Components](https://www.ibm.com/docs/en/zos-dce) for more information about the contents of this skeleton file. Refer to the [z/OS DCE Command Reference](https://www.ibm.com/docs/en/zos-dce) for a complete description of `uuidgen`.

Write the Interface Definition File

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

An interface definition file consists of two basic components:
• An interface header

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

• An interface body

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

The skeletal interface definition produced by the `uuidgen` utility provides an interface header that contains the newly generated UUID for the interface, a version number, and a dummy string `INTERFACENAME`. Replace this dummy string with the name of your interface, then add any additional interface header attributes your application requires (see the [z/OS DCE Application Development Guide: Core Components](#) for a complete description of interface header attributes).

The skeletal interface definition file also provides an interface body, which consists solely of an empty pair of braces (for example, `{}`). You fill in the space between the braces with your RPC interface's import, constant, type, and operation declarations, written in IDL. The [z/OS DCE Application Development Guide: Core Components](#) explains this process in more detail. In addition, consult the “Interface Definition Language” chapter of the same book for a complete description of the IDL syntax for specifying import, constant, type, and operation declarations.

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

### Write the Attribute Configuration File (ACF)

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

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

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

These binding methods are described in [Chapter 6, “Binding” on page 203](#). The binding method you choose determines how much attention your server's clients will have to devote to the upkeep of their binding handles.
The **z/OS DCE Application Development Guide: Core Components** provides a description of the ACF attributes available for use in attribute configuration files.

### Process the Files with the IDL Compiler

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

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

**Note:** Stub files used on the z/OS DCE platform must be generated with the z/OS DCE IDL compiler.

For further information on the IDL compiler, see the idl information in the **z/OS DCE Command Reference**.

### The Server Initialization

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

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

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

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

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

If dced did not start the server, it cannot control it. Therefore, one of the first things your server should do is to verify that dced started it by obtaining the configuration information:
Additional routines such as `dce_server_inq_uuids()` and `dce_server_inq_attr()` are also useful for obtaining information from `dced` about the running server.

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

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

**Set Up the Server's Objects**

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

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

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

DCE provides databases (known as a “backing stores”) to maintain typed data, like object UUIDs, between program invocations.

The following shows sample code to create UUIDs for server objects and how to store them using the Backing Store API (`dce_db*` routines):
Names are established so that applications can refer to objects in a way other than through the cumbersome UUID. Object UUIDs are generated in two ways:

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

After creating backing store headers (if desired) and opening the backing store databases, UUIDs are stored by calling the `dce_db_store_by_uuid()` routine. To store names associated with the UUIDs, call the `dce_db_store_by_name()` routine.
Object UUIDs in Bindings: Object UUIDs are often used in the DCE RPC binding mechanism. The details of RPC binding are explained later in the Section “Register the Server,” and more thoroughly in the Binding chapter of this guide. It all comes down to this: clients import only partial bindings from the namespace. These will carry them only as far as the endpoint mapper service of the dced on the destination server's host; it is dced's job to resolve the binding with a dynamic endpoint.

This means that some registration of bindings must be done by a server with the endpoint mapper. The minimum two items that have to be registered are interface UUIDs and bindings (the latter of which contains the server's dynamically allocated endpoints). With this information available, the endpoint mapper can inspect the incoming RPCs interface UUIDs, select one of the endpoints that was registered under them, and resolve the partial bindings. In addition, a server can register its object UUIDs with its endpoint mapper. This allows lookups of endpoints by object UUID rather than interface UUID; the advantage is that object UUIDs are much more specific than interface UUIDs, which may be registered by multiple servers at the same host.

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

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

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

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

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

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

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

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

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

\[
\text{dce_server_sec_begin(dce_server_c_login | dce_server_c_manage_key, \\ &status);}\
\]

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

Establish the Server Principal Identity

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

Manage the Server Key

When a server has its own identity, it takes on responsibility for the upkeep of its password using the Security Key Management routines.

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

To control access to the server's objects, Access Control List (ACL) managers are also set up.

Define Manager Entry Point Vectors for Each Set of Operations

“Manager” is the DCE term for the part of a server that actually implements a set of interface operations (the remote procedures), as distinguished from the more or less generic server initialization code described here. (See \text{sample_manager.c} in the \text{sample} application discussed in Appendix A, “A Sample Application” on page 271 for an example of manager code.) A Manager Entry Point Vector (EPV) is the data structure in which are recorded the entry addresses of the application routines that implement the server's operations, as offered through an interface. The server's stub code uses the EPV to dispatch incoming RPCs to the requested operations. For each interface the server supports, a default manager EPV is generated automatically by the IDL compiler. In order for the RPC runtime to properly dispatch remote procedure calls to the correct procedure, the server initialization code must declare the default EPVs and then register them with the runtime. For example,

\[
\text{extern rdaclif_v1_0_epv_t dce_acl_v1_0_epv;}\
\text{extern sample_bind_v1_0_epv_t sample_bind_epv;}\
\]

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

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

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

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

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

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

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

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

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

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

The `dce_server_register()` routine usually establishes all the services for a server at once. This is a reasonable approach for most applications, but some interfaces for services may have dependencies on the order in which they are enabled. After the server calls `dce_server_register()`, it can use a series of calls to `dce_server_disable_service()` and `dce_server_enable_service()` to disable and then later re-enable any interface offered by the server.
Tell RPC Runtime What Protocol Sequences to Use: The second thing registering the server does is it obtains a set of endpoints and associates them with the desired protocol sequences. Endpoints are the host's address numbers on which the server can receive incoming calls. This begins the process of actually setting up the information that the server's clients will need in order to bind to it. The endpoints are usually dynamically generated each time the server starts. However, some applications may use well-known endpoints that are the same every time the server starts. If well-known endpoints are used, they are typically defined in the interface definition with the `endpoint` attribute.

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

Register the Binding Information with the Endpoint Mapper Service: After server registration obtains the endpoints, the endpoints, protocol sequences, and object UUIDs are registered with the endpoint mapper service of the local host's `dced`.

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

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

An incoming RPC always has an interface UUID associated with it; therefore, if a server registers all of its endpoints with the interface it is offering, this will usually be sufficient for the endpoint mapper to send the incoming requests to one of the servers that offer the desired interface, even if there is more than one such server active on the machine. However, if the application is designed in such a way that the binding operation should not be generalized to the interface but must be made more specific (in other words, this server's clients should always bind to this server and no other, even if some other server happens to offer the same interface), then object UUIDs must be used to accomplish this. “Generic” interfaces offered by an application (such as the remote ACL or the DCE serviceability interface) require an object UUID in order to distinguish the application's “instance” of them; unique interfaces, however, do not require an object UUID.

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

- Endpoints
- Interface UUID

and optionally

- Object UUIDs

It then generates a cross-product table of all possible combinations of all values of the items. This allows it to find a valid endpoint for every possible valid object UUID/interface UUID combination.
The endpoint mapper is the first point of decision for an incoming RPC with a partial binding. The mapper makes its decision solely on the basis of the contents of its endpoint map. The object/type and manager EPV registrations that were done earlier have no effect on the endpoint mapper. Only after a client request arrives at the server does the server's runtime routines dispatch the request among multiple managers, if type managers have been registered by the server. The endpoint mapper knows nothing about registered object types.

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

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

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

**Specify Multithreadedness**

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

```c
if (pthread_create(&sigcatcher, 
    pthread_attr_default, 
    (pthread_startroutine_t)signal_handler, 
    (void*)0))
{
    dce_svc_printf(NO_SIGNAL_CATCHER_MSG);
    exit(1);
}
```

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

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

Although the execution threads are automatically managed by the RPC runtime, the developer is responsible for coding the manager routines according to thread-safe guidelines so that the threads will execute properly. For further information on thread-safe programming practices, see [Chapter 4, “Threads” on page 149](#).
Listen for Incoming Service Requests

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

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

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

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

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

Server operations can also be terminated by an exception or signal. DCE Threads defines all exceptions as “terminating,” which means that execution must be caught by an exception handler (if one exists) and then be resumed there, or the process will be terminated. Certain signals are defined by DCE Threads as exceptions, which means that these signals have the same general characteristics as exceptions. For more information on the DCE Threads exception handling interface, see Chapter 4, “Threads” on page 149.

Clean Up Code When the Server Terminates

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

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

```c
dce_server_unregister(server_uuid, &status);
```

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

If your application requires a partial shutdown or a particular order to the shutdown of services, you can use more specific routines such as `rpc_ep_unregister()` and `dce_server_disable_service()`.
**Unexport from the Namespace:** If the server is going to be out of service for an extended period, it should unexport any information it previously caused to be placed in the namespace. This will prevent future prospective clients from being misled into attempting to reach the server when it does not exist, and also will help to conserve resources in the namespace.

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

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

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

**Clean Up Security Information:** A call to the `dce_server_sec_begin()` routine should have a corresponding call to the `dce_server_sec_done()` routine to release resources allocated. In addition, your code should close any backing store databases used for ACL management by using `dce_db_close()`.

---

**The Client Binding and RPC Invocation**

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

The programmer designing clients must decide whether or not to use threads, and should have an understanding of multithreaded clients. DCE provides a set of tools for multithreaded programming; these are described in [Chapter 4, “Threads” on page 149](#).

**Import the Binding Information from the Namespace**

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

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

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

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

The normal way for a server to make its location known to clients is to export its binding information into the namespace. The client can then call the RPC name service library routines...
 rpc_ns_binding_import_begin(entry_name_syntax, entry_name,  
   if_handle, obj_uuid, &import_context, &status);

 rpc_ns_binding_import_next(import_context, &binding_handle,  
   &status);

 rpc_ns_binding_import_done(import_context, &status);

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

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

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

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

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

 rpc_binding_to_string_binding(binding_handle, &string_binding,  
   &status);

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

**Annotate the Binding Handle for Security**

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

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

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

- The server's principal name could be hardcoded in the client. This is not recommended practice for  
  reasons of robustness and flexibility.
- The client can be handed the name as input from the command line when it is invoked.
• The principal name can be the same as the name entry (binding information) name.

• The client can query the server’s principal name by calling `rpc_mgmt_inq_princ_name()`. It can then check group membership by calling `sec_rgy_pgo_is_member()`, using a known tested group.

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

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

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

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

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

**Setting Up for Authenticated RPC:** The client makes the following call in order to set up the security characteristics of the communications it is about to enter into with the server:

```c
    rpc_binding_set_auth_info(binding_handle, server_princ_name,
                               protect_level, authn_svc, login_context,
                               authz_svc, &status);
```

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

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

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

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

For further information about authenticated RPC, see the “Authentication” chapter of the [z/OS DCE Application Development Guide: Core Components](https://www.ibm.com).
Invoke Remote Procedure Calls

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

\[
\text{my_rpc_op(binding_handle, arg1, arg2, arg3)};
\]

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

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

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

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

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

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

Note that if you are using the auto-binding method and the binding becomes unusable for some reason, the RPC runtime will rebind under most conditions.
**The Result of Successful Binding:** If my_rpc_op() or its equivalent does succeed, the binding will as a result be complete (even if it was partial before), and the information in it can be regarded with much more assurance from then on. Subsequent remote procedure calls by the client to the same server will go straight to the bound-to server.

**The Server's Manager of RPC Requests**

As was explained, server threads are automatically spawned by the RPC runtime in the server manager to handle incoming remote procedure calls from clients. The number of calls that can be concurrently handled depends on the value of the \textit{max\_calls\_exec} parameter specified in the call to \texttt{rpc\_server\_listen()}. The thread is created by the RPC runtime and begins execution in the operation requested. When the operation is completed, the thread is automatically terminated (by the RPC runtime).

See also the “Programming with Threads” chapter of the \textit{z/OS DCE Application Development Guide: Core Components} and the “Threads” chapter of the \textit{z/OS DCE Application Development Reference} for a comprehensive discussion of DCE Threads.

**Get the Client's Credentials**

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

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

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

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

\begin{verbatim}
rpc\_binding\_inq\_auth\_caller(binding\_handle, privileges, server\_princ\_name, protect\_level, authn\_svc, authz\_svc, &status);
\end{verbatim}

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

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

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

**Get RACF Authorization using RACF-DCE Interoperability**

You may want to use the authorization and auditing capabilities provided by Resource Access Control Facility (RACF) for the server portion of a client-server application that resides on z/OS. With z/OS DCE, this is possible using the information created when z/OS users are enrolled in the RACF-DCE interoperability feature. For information on enrolling in and using RACF-DCE interoperability, see [z/OS DCE Administration Guide](#).

**Get the Object's Access Control List (ACL)**

This step is reached if the client requests access to any object, resource, or service that is managed by the server, to which ACLs are attached. As previously mentioned, the application must implement its own ACL manager if it wants to use ACLs to control access to its resources. For further details on how to go about creating an ACL manager, see "RPC Threads" on page 162.

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

For the client, all that is necessary is to possess a binding to the object whose ACL is to be operated on. As long as the application exposes the resources it manages as accessible objects (via the namespace), then the DCE ACL interface provides for a client's being able to bind to the object by calling `sec_acl_bind()`. (In fact, this kind of object-oriented binding model can be very useful, and is discussed in further detail in Chapter 6, "Binding" on page 203.) Note that the `sec_acl_...()` routines use an "ACL handle" to specify the object whose ACL is to be accessed, so `sec_acl_bind()` must always be called to obtain this handle, even if the client is already bound to the object's server.

There is a user interface into the ACL operations, embodied in the `dcecp acl` command. For further information, see the [z/OS DCE Command Reference](#).

Server applications can use the DCE ACL library routines to implement ACL managers. The DCE ACL library is an implementation of the remote ACL (`rdacl`) interface, designed in such a way as to allow any DCE application to use it instead of having to implement the interface itself. In DCE 1.0, applications that wished to use the DCE ACL functionality had to implement the full remote interface themselves; in DCE 1.1 this is no longer necessary. For further information, see Chapter 5, "Security" on page 163.

**Make the Authorization Decision**

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

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

Although it is up to the application to implement its own ACL storage, testing algorithms and manager types, there are certain DCE-wide design conventions that should be kept in mind and departed from only for good reason. Among these are the following:
• Standard DCE ACL entry types: the kinds of entries that can occur in an Access Control List (for example, **user**, **group**, and so on).

• Standard privileges: the kinds of access that a principal can have to a protected object (for example, read, write, and so on).

• Standard inheritance rules: these rules govern the default characteristics of ACLs created for newly created objects.

• Standard access algorithm: the order in which a client's credentials are matched against the various possible entry types.

Information about these topics for application developers designing their own ACL model can be found in the “Access Control List Application Programming Interface” chapter of the [z/OS DCE Application Development Guide: Core Components](#), where all the DCE authorization conventions are described in detail.

**Service the RPC Request**

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

Note that this step and the steps involving getting the object's ACL and making the authorization decision are somewhat intertwined. Something like the following could occur:

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

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

3. If the client is found to have the requisite privilege, **my_rpc_op()** proceeds.

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

**Return the Results and Resume Listening**

At the completion of the operation, the RPC thread that was automatically spawned to execute it is terminated by the RPC runtime.

From the server's point of view, the result of completing the remotely called routine is that it reenters the "listen" loop, waiting for further remote calls. The server's runtime handles all the communications details of actually sending any requested data to the client. As far as the server is concerned, it is still blocking on the call to **rpc_server_listen()** which was made earlier. If **max_calls_exec** was specified to be greater than 1 in that call, other threads may still be executing at this time in response to other requests that have been received from other clients. In any case, the call to **rpc_server_listen()** will not return until one of the server's own management routines, or a client, makes a successful call to **rpc_mgmt_stop_server_listening()**. If this happens, the RPC runtime will stop accepting incoming client requests to the server. When all the currently executing operations have been completed, the call to **rpc_server_listen()** will return.

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

From the client's point of view, the server's return at the end of its remotely called routine results in the client's returning from a seemingly locally executed routine.
Continue: The client now goes on about its business, which may include performing other remote procedure calls.

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

**Writing a Simple Distributed Application on z/OS**

As an entry point to creating actual distributed applications on the z/OS operating system, this “how-to” section guides you through the application development steps talked about earlier for a simple DCE application using z/OS DCE.

The following runtime environments are addressed in each step:

- z/OS UNIX System Services shell
- z/OS Time Sharing Option (TSO/E)
- Batch.

Each step describes the action you take in the above environments. If a particular environment does not apply, skip that section.

Tools are available with z/OS DCE to assist you in writing your applications. This chapter introduces these tools and guides you through their use in the Shell, TSO/E, and TSO/E batch environments. In the TSO/E environment, you can invoke these tools in both batch mode and foreground mode. For batch mode, use the example JCL contained in this chapter to write and submit your own applications.

**High-Level Application Development Steps**

This section describes the major steps you follow to create and run your distributed application in the three z/OS runtime environments. In the Shell environment, the main tools for DCE application development are the `uuidgen` facility, the `idl` compiler, the `c89` compiler, and the `make` facility. In the TSO/E environment, the main tools are the following REXX EXECs: UUIDGEN and IDL, and Command lists (CLIST): CC and CPLINK. In the batch environment, the main tools are the following Cataloged Procedures (PROC): UUIDGEN, IDL, EDCC and EDCPL. In the Shell, you generally use Hierarchical File System (HFS) files for your applications. In the TSO/E and batch environments, you generally use partitioned data set (PDS) files or physical sequential datasets.

Some of the more complicated steps and options have been purposefully left out to simplify the process.

1. Creating files for your application
2. Generating a *universal unique identifier* (UUID) and IDL template file
3. Naming the interface for your application
4. Defining the interface operations
5. Compiling the interface with the IDL compiler
6. Writing the server and manager code for your application
7. Writing the client code for your application
8. Compiling the client and server programs
9. Link-editing your application with the DCE, TCP/IP, and C/C++ runtime libraries
10. Building your DCE application.
Notes:

1. Your IDL file can only be in code page IBM-1047. If you port your IDL file from another platform, ensure that it is converted to code page IBM-1047. You can use the `iconv` command to do this conversion. For more information about `iconv`, see the [z/OS UNIX System Services Command Reference](https://publib.boulder.ibm.com/infocenter/zos/v2r11/index.jsp?topic=%2Fcom.ibm.zos.v2r11.cmd%2Ficonv.htm). Note that the stubs that are output from the IDL compiler are also in code page IBM-1047.

2. In TSO/E and batch environments, the length of the entire information passed using the PARMS parameter cannot exceed 100 characters. This number includes any commas that are passed to the program but excludes enclosing parentheses or apostrophes that are not passed. For further information on the syntax of the PARMS parameter, consult [z/OS MVS JCL Reference](https://publib.boulder.ibm.com/infocenter/collection/predix_zos_mvs_jcl_reference_main.html). SA22-7597.

3. References are made throughout this chapter to ISPF and SDSF panels and tools to help you carry out the application tasks. Some of these panels may appear to be different on your system, or you may not be authorized to use certain ones. If you have problems, consult your z/OS Systems Programmer for alternative methods to accomplish the same task.

Step 1. Creating Files for Your Application

For DCE applications, you typically code the five files contained in Table 1. It lists the HFS files names and the corresponding PDS file member names. In this book, HFS files and associated directory are presented in lowercase, while any dataset names and members are presented in uppercase. The name of your DCE application is represented by `applnm` for HFS files and `APPLNM` for PDS files.

For your partitioned data sets, `APPLNM` cannot be greater than six characters in length.

You do not have to use the naming convention presented in this table for your application files. The names in the examples are for illustration only. You can generally access either the HFS file or the PDS file when you use the DCE utilities in the Shell environment. You can only access PDS files when you use the DCE utilities in the batch or TSO/E environments. You can copy the contents of an HFS file into a PDS by using the TSO/E OGET command, and you can copy the contents of a PDS file into an HFS file using the TSO/E OPUT command. Refer to [z/OS UNIX System Services User's Guide](https://publib.boulder.ibm.com/infocenter/zos/v2r11/index.jsp?topic=%2Fcom.ibm.zos.v2r11.cmd%2Fos Unix.html) for more information on these commands.

**Note:** In the table below, `USERPRFX` represents your z/OS logon identification, and `APPLNM` represents the name of your application.

### Table 1 (Page 1 of 2). User-Written Files for DCE Applications

<table>
<thead>
<tr>
<th>File</th>
<th>Hierarchical File System</th>
<th>Partitioned Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL file¹</td>
<td>File name: <code>applnm.idl</code></td>
<td><code>USERPRFX.IDL(APPLNM)</code></td>
</tr>
<tr>
<td>C source files²</td>
<td>File names:</td>
<td></td>
</tr>
<tr>
<td>• Server</td>
<td><code>applnm_server.c</code></td>
<td><code>USERPRFX.C(APPLNMSR)</code></td>
</tr>
<tr>
<td>• Manager</td>
<td><code>applnm_manager.c</code></td>
<td><code>USERPRFX.C(APPLNMMR)</code></td>
</tr>
<tr>
<td>• Client</td>
<td><code>applnm_client.c</code></td>
<td><code>USERPRFX.C(APPLNMCCL)</code></td>
</tr>
</tbody>
</table>
Table 1 (Page 2 of 2). User-Written Files for DCE Applications

<table>
<thead>
<tr>
<th>File</th>
<th>Hierarchical File System</th>
<th>Partitioned Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACF file³</td>
<td>File Name: applnm.act</td>
<td>USERPRFX.ACF(APPLNM)</td>
</tr>
</tbody>
</table>

Purpose:

1. There is usually one IDL file for a simple DCE client and server application. You write the IDL file using the **Interface Definition Language (IDL)**. The IDL file defines all aspects of an interface that affect data passed over the network between a DCE client and the server. You write the IDL file for your application in "Step 4. Defining the Interface Operations" on page 37.


3. This optional file is used to code the **Attribute Configuration Language** that modifies the interaction between application code and stubs, and to declare certain DCE attributes.

You can have more than one member in any of the table’s partitioned data sets. For example, you can have multiple members in the IDL file representing many interface definitions, depending on your application’s requirements. You can also write additional header members that contain special definitions for your application. For simple DCE applications, typically only one member is required.

During the creation of a DCE application, the files listed in the following table are generated by z/OS DCE utilities such as the IDL compiler or the c89 compiler. If you use the naming convention of Table 1 on page 30, you will generate most of the files contained in the table, depending on the options you choose when you run the z/OS DCE utilities.

**Note:** The z/OS DCE IDL compiler no longer automatically generates auxiliary files. If you are using a makefile that contains dependencies on auxiliary files, dummy files can be created by defining environment variable `IDL_GEN_AUX_FILES`.

Table 2 (Page 1 of 2). System-generated Files for DCE Applications

<table>
<thead>
<tr>
<th>File</th>
<th>Hierarchical File System</th>
<th>Partitioned Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stub files¹</td>
<td>File names:</td>
<td></td>
</tr>
<tr>
<td>• Server</td>
<td>applnm_sstub.c</td>
<td>USERPRFX.C(APPLNMSS)</td>
</tr>
<tr>
<td>• Client</td>
<td>applnm_csstub.c</td>
<td>USERPRFX.C(APPLNMC)</td>
</tr>
<tr>
<td>Header file²</td>
<td>File name: applnm.h</td>
<td>USERPRFX.H(APPLNM)</td>
</tr>
<tr>
<td>Object files³</td>
<td>File names:</td>
<td></td>
</tr>
<tr>
<td>• Server</td>
<td>applnm_server.o</td>
<td>USERPRFX.OBJ(APPLNMSSR)</td>
</tr>
<tr>
<td>• Manager</td>
<td>applnm_manager.o</td>
<td>USERPRFX.OBJ(APPLNMMR)</td>
</tr>
<tr>
<td>• Client</td>
<td>applnm_client.o</td>
<td>USERPRFX.OBJ(APPLNMC)</td>
</tr>
</tbody>
</table>
Table 2 (Page 2 of 2). System-generated Files for DCE Applications

<table>
<thead>
<tr>
<th>File</th>
<th>Hierarchical File System</th>
<th>Partitioned Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load file4</td>
<td></td>
<td>USERPRFX.LOAD(APPLNM.SR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USERPRFX.LOAD(APPLNM.CL)</td>
</tr>
</tbody>
</table>

Purpose:

1. The stub files, one for your server and one for your client application, contain the code for marshalling and unmarshalling data, message handling, and other details of network communication. The stubs are generated when you compile the IDL file in “Step 5. Compiling the Interface with the IDL Compiler” on page 37. Depending on the IDL compiler options you choose, you can generate the stubs directly as object code instead of C code.

2. The header file is common across your server and client application. It is obtained from compiling the IDL file in “Step 5. Compiling the Interface with the IDL Compiler” on page 37. The header file contains declarations and definitions that are for general use in an application. You must compile the header along with the source code when compiling either your client or server application.

3. The object files contain the object code for your application that is obtained from the compile step in “Step 8. Compiling the Client and Server Programs” on page 40.

4. The load files contain the client and server application load modules that are created in “Step 9. Link-Editing Your Application” on page 45 so you can run your application.

In the Shell: To enter the Shell, enter OMVS on any TSO/E command line. You do not need to allocate any hierarchical file system (HFS) files for your DCE applications, but for your convenience, you may want to make a directory where you can store all your associated DCE application files. The HFS files are created for you when you first edit them, using the TSO/E OEDIT command. You might also want to store all files that you generate using z/OS DCE utilities in this directory.

Following are the HFS file extensions (in bold) that are appended to your application name by the z/OS DCE utilities:

<table>
<thead>
<tr>
<th>File</th>
<th>File extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>header file</td>
<td>applnm.h</td>
</tr>
<tr>
<td>server stub</td>
<td>applnm_sstub.c</td>
</tr>
<tr>
<td>client stub</td>
<td>applnm_cstub.c</td>
</tr>
</tbody>
</table>

There is no restriction on the length of the IDL file name denoted by applnm if it is an HFS file, as there is for PDS files.

In TSO/E and Batch: Set up your environment by allocating all the necessary partitioned data sets for your application. Seven data sets are required:

- USERPRFX.JCL
- USERPRFX.IDL
- USERPRFX.C
- USERPRFX.H
- USERPRFX.OBJ
- USERPRFX.LOAD
- USERPRFX.ACF

32 DCE Application Development Guide: Introduction and Style
Notes:

1. You do not have to use the above naming convention for your application data sets. The names in the examples are for illustration only.

2. The application will be members of these allocated partitioned data sets and their names cannot be greater than six characters in length.

The following table shows example attributes for the above data sets. You should adhere to the record format and record lengths presented, or you may encounter difficulty running the JCL. You can adjust the block sizes of your data sets as required for your needs (in multiples of the record length).

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Record Format</th>
<th>Record Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>USERPRFX.JCL (Note 1.)</td>
<td>FB</td>
<td>80</td>
</tr>
<tr>
<td>USERPRFXIDL</td>
<td>FB</td>
<td>80</td>
</tr>
<tr>
<td>USERPRFX.C</td>
<td>FB</td>
<td>80</td>
</tr>
<tr>
<td>USERPRFX.H</td>
<td>FB</td>
<td>80</td>
</tr>
<tr>
<td>USERPRFX.OBJ</td>
<td>FB</td>
<td>80</td>
</tr>
<tr>
<td>USERPRFX_LOAD</td>
<td>U</td>
<td>6160</td>
</tr>
<tr>
<td>USERPRFX.ACF</td>
<td>FB</td>
<td>80</td>
</tr>
</tbody>
</table>

Notes:

1. z/OS DCE does not require any particular JCL attributes. These values are given as an example only.

You can allocate the above data sets manually using the 'Allocate new data set' panel (option 3.2 in the ISPF utilities) or however you normally do it in TSO/E. For your convenience, z/OS DCE supplies an IDLALLOC REXX EXEC that allocates the necessary data sets required for a DCE application. Note that IDLALLOC only allocates data sets that do not already exist.

From TSO/E use the command:

`idalloc`

IDLALLOC will present a panel which includes a DSN PREFIX field allowing for Data Set Name customization. See Figure 2 on page 34.

The data sets created will be organized as Partitioned Organization (PO). Table 3 describes the data sets created using IDLALLOC. Verify that your data sets conform to these characteristics by using the 'Data set information' utility before proceeding. When you invoke the IDLALLOC utility, you will see a menu similar to the one contained in Figure 2 on page 34.
Figure 2. IDLALLOC Utility Menu

You can adjust any of the data set characteristics for your own requirements. You should change the default values for block size within the IDLALLOC REXX EXEC to optimize the block size to your system’s DASD devices. If you need the CACF, CIDL, TACF, CTDL, and CTCF files to be allocated, make sure primary quantity and secondary quantity are set to a number other than zero.

Step 2. Generating a UUID and IDL File

Next, you create a Universal Unique Identifier (UUID) to uniquely label your application’s interface. Through this identifier, the DCE naming service identifies and locates your application server throughout DCE. The UUID is generated along with a template for your application’s IDL file when you run the UUID generator (UUIDGEN) that is available in all three run time environments. Figure 3 shows an example of an IDL file that is generated using UUIDGEN. The contents of the file are the same in any of the three environments, except for the UUID generated.

```
[ uuid(20818313-4143-19ea-a6e0-0000dce12345),
  version(1.0) ]
interface INTERFACENAME
{
}
```

Figure 3. Example of an IDL Template

The 128-bit UUID, in parentheses, is a string of multiple fields of hexadecimal characters separated by hyphens. Each field has a fixed length. The UUID that you generate is similar to the one presented in the above example, but is unique for your application across DCE.
**In the Shell:** Use the example command in Figure 4 to generate a UUID for your application. In addition to generating a UUID, entering this command creates a template for your application’s IDL file.

```bash
uuidgen -i -o <dir_name/applnm>.idl
```

*Figure 4. Sample Shell Command to Run the UUID Generator*

In the above example, the `uuidgen` command is run with the `-i` option, which specifies that a skeletal interface definition file is to be generated that includes the UUID. The `-o` option redirects the output from your screen to the file specified by `<dir_name/applnm>.idl`. If you do not specify a directory name, the file `applnm.idl` is placed in your current directory. For a complete description of the input parameters for the UUID generator in the Shell, see [z/OS DCE Application Development Reference](#).

As an example, to create an IDL file for an application called *stock* in your current directory, use the command shown in Figure 5.

```bash
uuidgen -i -o stock.idl
```

*Figure 5. Sample Shell Command to Create a UUID for STOCK Application*

To display the contents of the resulting `stock.idl` file, enter the following command:

```bash
cat stock.idl
```

The contents of the `stock.idl` file resembles the IDL file in Figure 3 on page 34 except that it contains a unique UUID.

**Using a Partitioned Data Set:** To create your IDL file in a preallocated partitioned data set, replace `<dir_path_name/applnm>.idl` in the example given in Figure 4 with the fully qualified data set name preceded by `//` and enclosed in double quotation marks. The `uuidgen` example then becomes:

```bash
uuidgen -i -o "//USERPRFX.IDL(STOCK)"
```

*Figure 6. Sample Shell Command to Create UUID Using a PDS*

The contents of the `USERPRFX.IDL(STOCK)` data set resembles the IDL file in Figure 3 on page 34 except it contains a unique UUID.

**In TSO/E:** You can also generate a UUID in foreground mode by entering the following TSO/E command from any ISPF command line:

```tso
tso uuidgen uuidgen parameters
```

To create an IDL file for an application called *stock*, use the command shown in Figure 7.

```bash
uuidgen -i -o "//USERPRFX.IDL(STOCK)"
```

*Figure 7. Sample TSO/E Command to Create a UUID for STOCK Application*

The IDL file created and stored in `USERPRFX.IDL(STOCK)` resembles the IDL file in Figure 3 on page 34 except it contains a unique UUID.
**Using an HFS File:** To create your IDL file as an HFS file, replace "/"'USERPRFX.IDL(APPLNM)'" with the fully qualified path name for your HFS file. If you do not specify a path, the HFS file is created in your home directory.

The stock IDL file example then becomes:

```bash
cmd uuidgen -i -o <your_path>/stock.idl
```

*Figure 8. Sample TSO/E Command to Create a UUID using an HFS File*

**In Batch:** Use the example JCL contained in Figure 9 to generate a UUID for your application. In addition to generating a UUID for your application, running this JCL creates a template for your application’s IDL file.

```bash
//JOBNAME JOB (ACCOUNT)...your_job_parameters
//*******************************************************************************************
/* JCL to run UUIDGEN */
//*******************************************************************************************
//UUIDGEN EXEC UUIDGEN,
// PARMS='your_uuidgen_parameters'
```

*Figure 9. Sample JCL to Run the UUID Generator*

Pass any input parameters for the UUID generator using the PARMS statement in the JCL. See [Z/OS DCE Application Development Reference](#) for information on the input options when running the UUID generator.

For example, to create an IDL file for an application called STOCK, use the JCL contained in Figure 10.

```bash
//JOBNAME JOB (ACCOUNT)...your_job_parameters
//*******************************************************************************************
/* JCL to run UUIDGEN */
//*******************************************************************************************
//UUIDGEN EXEC UUIDGEN,
// PARMS='-i -o '//STOCK.IDL(STOCK)''
```

*Figure 10. Sample JCL to Create a UUID for STOCK Application*

The IDL file created and stored in USERPRFX.IDL(STOCK) resembles the IDL file in Figure 3 on page 34 except it contains a unique UUID.

**Using an HFS File:** To create your IDL file as an HFS file, replace "/"'APPLNM.IDL(APPLNM)'" with the fully qualified path name for your HFS file. If you do not specify a path, the HFS file is created in your home directory.

The JCL to generate the stock IDL file then becomes:

```bash
//JOBNAME JOB (ACCOUNT)...your_job_parameters
//*******************************************************************************************
/* JCL to run UUIDGEN */
//*******************************************************************************************
//UUIDGEN EXEC UUIDGEN,
// PARMS='-i -o '<your_path>/stock.idl',
```

*Figure 11. Sample JCL to Create UUID Using an HFS file*
Step 3. Naming the Interface for Your Application.

This is the first step in customizing the IDL file created for your application. Simply replace the string `INTERFACENAME` with the name you want for your application’s interface. This step is common across all three z/OS UNIX environments.

The characters that you may use to make up an interface name include the following:

- Alphabetic characters, A through Z (uppercase)
- Alphabetic characters, a through z (lowercase)
- Numeric characters, 0 through 9
- Underscore symbol, (_).

The length of the interface name is a maximum of 17 characters. Because IDL is case sensitive, you can use lowercase and uppercase characters to represent different interfaces. The IDL compiler rejects any interface names that do not conform to the above rules.

Step 4. Defining the Interface Operations

Define the operation comprising the interface within the braces and thus define the operation of your server application. This task includes naming the operation, defining the return values and arguments passed to your server from your client, and defining the arguments returned to your client from your server. For more information on writing your interface operations, refer to the “Operation Declaration” section of the [z/OS DCE Application Development Guide: Core Components](#). This step is common across all three z/OS UNIX environments.

Step 5. Compiling the Interface with the IDL Compiler

Upon completing the IDL file for your application, you can compile it using the IDL compiler. The IDL compiler normally outputs three files and two optional files depending on the contents of your IDL file and compiler options you choose. All these files are compiled with your application source code:

<table>
<thead>
<tr>
<th>File</th>
<th>Hierarchical File System File Name</th>
<th>Partitioned Data Set Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>header</td>
<td>applnm.h</td>
<td>USERPRFX.H(APPLNM)</td>
</tr>
<tr>
<td>client stub</td>
<td>applnm_cstub</td>
<td>USERPRFX.C(APPLNMCS)</td>
</tr>
<tr>
<td>server stub</td>
<td>applnm_sstub</td>
<td>USERPRFX.C(APPLNMS)</td>
</tr>
</tbody>
</table>

Notes:

1. For PDS names, `APPLNM` cannot be more than six characters long, or the IDL compiler will reject it. There is no similar restriction on HFS IDL file names.

The header contains the definitions and declarations derived from the input IDL file that are for general use in the development source code. The stubs, one for your server and one for your client, contain the code for marshalling and unmarshalling data, message handling, and other details of network communications management.

The IDL compiler supplied with z/OS DCE uses an input IDL file to define a set of subroutine interfaces. Each set of interfaces is identified by its Universal Unique Identifier (UUID). Along with the IDL file is the Attribute Configuration File (ACF) that controls the IDL compiler’s interpretation of the IDL file. With the ACF, you can customize the client or server stubs to suit your local environment. For the exact syntax and available options for running the IDL compiler, refer to [z/OS DCE Application Development Reference](#).
When you ship a server to your users or customers, you must ship the IDL and ACF files as well. The
users can regenerate the client and server stubs after modifying the ACF to suit their local environment.
You do not have to re-create the stubs if customization is not required.

**In the Shell:** After you have written your interface definition, use the sample command shown in
Figure 12 to compile the IDL file for your application, producing the header and stub files. This command
does not invoke the C preprocessor and the C/C++ compiler. It will generate the stubs for the client and
server.

```
idl applnm.idl -no_cpp -keep c_source
```

*Figure 12. Sample Shell Command to Run the IDL Compiler*

The IDL compiler can also produce stubs in object format by invoking the C/C++ compiler. In this case,
the `.c` portion of the output file is replaced by `.o`.

**Using a PDS File:** To create the stubs and header output into a preallocated PDS file and if your IDL
source file is an HFS file, use the following Shell command:

```
idl applnm.idl -no_cpp -keep c_source -out "//'<DSNAME>">
```

*Figure 13. Sample Shell Command to Run the IDL Compiler — HFS File*

If your IDL source file is a PDS file, use the following Shell command:

```
idl "//'<USERPRFX.IDL(<IDLFILE>)'" -no_cpp -keep c_source -out "//'<DSNAME>">
```

*Figure 14. Sample Shell Command to Run the IDL Compiler — PDS File*

To run the IDL compiler, the environment variable **NLSPATH** must be set so that the compiler can find its
catalog of diagnostic messages. The value of this variable must include the string `dir/%N`, where `dir` is the
directory path in which the `dceidl.cat` file resides. When the LANG environment variable is set, you can
set NLSPATH to `/usr/lib/nls/msg/%L/%N`. The value of the LANG environment variable is substituted for
the `%L` in the directory path. For example:

```
export NLSPATH=/usr/lib/nls/msg/En_US.IBM-1047/%N
```

Note that the `%N` in the example is a variable that represents the name of a catalog file. It is not a
directory.

The IDL compiler offers many options that enable you to:
- Invoke the C/C++ compiler and/or preprocessor, if desired
- Specify which directories are searched for imported files
- Select the output files that are generated
- Specify how the output files are named

When you compile the definition of a remote interface, you must ensure that the system IDL directory is
among those that the IDL compiler searches when it searches for imported files, because any remote
interface implicitly imports `nbase.idl`. 
In TSO/E:  Use the sample TSO/E EXEC, IDL, shown in Figure 15 to compile the IDL file for your application and produce the files listed above. You can specify the prefix for your IDL file using the -userpfx option as shown below. Note that the IDL EXEC handles PDSs only.

```
idl APPLNM -keep c_source -no_cpp -userpfx USERPRFX.APPLNM
```

*Figure 15. Sample TSO/E Command to Run the IDL Compiler*

In Batch:  Use the sample JCL shown in Figure 16 to compile the IDL file for your application and produce the files listed above. The PROC that runs is IDL. Pass any parameters to the IDL compiler using the `PARM='idl_compiler_options'` statement. Note that IDL handles PDSs only.

```
//JOBNAME JOB (ACCOUNT)...your_job_parameters
//******************************************************************************
//* RUN THE IDL COMPILER *
//******************************************************************************
//IDLCOMP EXEC IDL,USERPFX='USERPRFX',
// PARM='GREET -keep c_source -no_cpp'
```

*Figure 16. Sample JCL to Run the IDL Compiler in Batch*

**Step 6. Writing Your Server and Manager Code**

The server is the first of three C source code files that you write for your application. The server program is really broken into two parts: the main server routine and the manager routine. These two programs could be combined; however, the convention is to split them as follows:

- The server routine registers its interface with the RPC runtime, and obtains and exports its binding information. It contains the generic portion of the server code that initializes the server as a whole to the DCE system and sets up the server to listen for incoming requests from clients. The generic RPC functions that the server code likely performs in a simple application are:

<table>
<thead>
<tr>
<th>RPC function call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rpc_server_register_if(...)</code></td>
<td>Registers at least one interface.</td>
</tr>
<tr>
<td><code>rpc_server_use_all_protseqs(...)</code></td>
<td>Tells the RPC runtime to use all supported protocol sequences for receiving remote procedure calls.</td>
</tr>
<tr>
<td><code>rpc_ep_register(...)</code></td>
<td>Registers endpoints.</td>
</tr>
<tr>
<td><code>rpc_server_listen(...)</code></td>
<td>Listens for remote procedure calls.</td>
</tr>
</tbody>
</table>

After you perform the above initialization tasks, your server is prepared to receive remote procedure calls from clients.

- The manager routine is the portion of the server that actually carries out the set of operations that you want the server to perform.

**Note:** On MVS, multithreading in DCE applications usually requires less *below-the-line* stack memory than the default provided by z/OS Language Environment. Thus, for any multithreaded DCE application, it is recommended that you reset the stack size. DCE server applications are inherently multithreaded. DCE client applications that use `pthread_create()` are also multithreaded and require that you reset the stack size.

The stacksize can be reset by adding the following `#pragma` directive in your code prior to any C source statements, except other `#pragmas`:
Step 7. Writing the Client Code for Your Application

The client code makes the function call to your server. The client obtains the binding information for the server from the Directory. Once such items as the correct protocol sequence, the binding handles, and so on are received from the DCE runtime to establish communications with the server, your client makes the function call, passing along any required input parameters. Once your client receives the results of the function call from the server, it may process the parameters received or make additional function calls before ending.

Step 8. Compiling the Client and Server Programs

Once you have written the client, server, and manager source code, you are ready to compile them along with the client and server stubs using the C/C++ compiler. To compile your DCE application, you use the **c89** compiler, the interface to the C/C++ compiler for programs compiled in the shell. In TSO/E, use the CC EXEC, and in batch use the EDCC PROC; both are supplied by the z/OS C/C++ product.

For DCE applications, you specify the following definitions using the **DEFINE** C/C++ compiler option, in all three environments:

- _DCE_THREADS
- _OPEN_SYS
- MVS.

In all three environments the following compiler option is recommended when compiling C source that contains references to functions or external variables imported from the DCE Dynamic Link Library:

- DLL

In the TSO/E or batch environments, specify the following minimum set of C/C++ compiler options, otherwise the compile or link-edit step may fail:

- LONGNAME (Long name support)
- NOMAR (No margin)
- NOSEQ (No sequence).

You may find the SOURCE and SHOWINC options to be useful for debugging compile errors in your source code.

You can compile your source files individually if you want, or you can combine the compile and link steps into one step as in "Step 10. Building Your DCE Application" on page 48.

z/OS DCE uses the Dynamic Link Library support of the z/OS C/C++ compiler. As a result, some new parameters are required when DCE applications are compiled and link-edited.

In order to access functions and external variables exported by a DLL, a C program is required to be compiled with the DLL compiler option. It is also required that the definition side-deck that is associated with the DLL be included when a program is link-edited. For a more complete discussion of the DLL support provided by z/OS refer to the z/OS Language Environment Programming Guide.
There are three types of DCE applications that are affected by this change:

- Existing DCE applications that do not require any updates or maintenance.
- Existing application that will require updates or maintenance.
- New applications that will be developed on or ported to z/OS.

Existing applications that do not require any updates will continue to run using the new DCE DLL. Recompiling or relink-editing of existing z/OS DCE application modules is not required.

Existing applications that need updating have a choice of compiling the changed C source with or without the DLL compiler option. This is provided for compatibility. In either case, when these applications are link-edited they will have to include the definition side-deck associated with the DCE DLL. The definition side-deck for the DCE DLL is installed as the HFS resident file `/usr/lib/EUVPDLL.x` and the PDS resident file `SEUVEXP(EUVPDLL)`. The PDS may have installation-dependent high-level qualifiers prepended to it.

New DCE applications being developed on or ported to z/OS whose C source parts reference functions or external variables exported by the DCE DLL should be compiled with the DLL compiler option. When these applications are link-edited they will have to include the definition side-deck that is associated with the DCE DLL. The examples in this book reflect the changes to the compiling and link-editing of DCE applications caused by the modification of the DCE DLL.

**In the Shell:** You can use any of `c89` compiler options to compile your DCE application. For a complete list of the options, and information on how you set them using the `#pragma`, `#define` preprocessor directives or interactively using the `c89` utility, refer to the [z/OS UNIX System Services Command Reference](https://www.ibm.com/support/knowledgecenter/SSEPGG_2.4.0/com.ibm.zos.v2r4.com.ibm.zos.command.ref/zoscomref.html). For DCE applications, you must specify the directory where the DCE header files specified in the `#include` directive in your source are found using the `-I` option. DCE header files are found in the `/usr/include` and `/usr/include/dce` directories. They are described in the appendix of [z/OS DCE Application Development Reference](https://www.ibm.com/support/knowledgecenter/SSEPGG_2.4.0/com.ibm.zos.v2r4.com.ibm.zos.dce.development/rdfch401.html).

**Note:** If your DCE source is an HFS file, the object file is created as an HFS file in the working directory. If your DCE source is in a PDS, the object file is created as a PDS and the object file is placed in a data set with the qualified name of the source and identified with OBJ as a qualifier.

**Compiling Your Client using HFS Files:** Following is an example command to compile the C source for the client portion of your DCE application using HFS files. Note that you can add any `c89` compiler options that you require.

```
c89 -c -DMVS -D_DCE_THREADS -D_OPEN_SYS -W0,DLL applnm_client.c applnm_cstub.c
```

*Figure 17. Compiling Your Client using HFS Files*

**Compiling Your Server using HFS Files:** Following is an example command to compile the C source for the server portion of your DCE application using HFS files.

```
c89 -c -DMVS -D_DCE_THREADS -D_OPEN_SYS -W0,DLL applnm_server.c \ applnm_sstub.c applnm_manager.c
```

*Figure 18. Compiling Your Server using HFS Files*

The above commands will create object files in `your_dir` with the same name as the C source file but with the `c` file extension replaced by the `o` extension.
Compiling Your Client using PDS: Following is an example command to compile the C source for the client portion of your DCE application using PDS files.

```c
cc89 -c -DMVS -D_DCE_THREADS -D_OPEN_SYS -wo,DLL \
""/"'USERPRFX.C(APPLNMCL)"' \
""/"'USERPRFX.C(APPLNMCS)"'
```

Figure 19. Compiling Your Client using PDS Files

Compiling Your Server using PDS: Following is an example command to compile the C source for the server portion of your DCE application using PDS files.

```c
cc89 -c -DMVS -D_DCE_THREADS -D_OPEN_SYS -wo,DLL \
""/"'USERPRFX.C(APPLNMSR)"' \
""/"'USERPRFX.C(APPLNMSS)"' \
""/"'USERPRFX.C(APPLNMRS)"
```

Figure 20. Compiling Your Server using PDS Files

As you can see there are many combinations of HFS files and PDS files that can be used. You may find it easier to work with either data sets or HFS files and mix-and-match the files only when necessary.

In TSO/E: You can compile your DCE applications in the TSO/E environment using the CC EXEC supplied by the z/OS Language Environment product. You may need to customize some of the data sets required by the CC EXEC for your environment. Refer to the z/OS C/C++ User's Guide for information on how to use the CC EXEC.

Note: If input parameters to the CC EXEC are all capitalized, C/C++ converts these parameters to lowercase by default. For DCE applications, you are required to specify certain compiler options in uppercase. To enable the CC EXEC to preserve the case of your input parameters, you need to pass at least one parameter to the C/C++ compiler in mixed case.

If the C/C++ SCEERUN library is not in your Link Pack Area (LPA), your Link List (LNKLST), or your STEPLIB, see your System Programmer to ensure the C/C++ SCEERUN modules are available to your TSO/E session.

Following is an example of compiling the client stub of your DCE application:

```c
cc APPLNM.C(APPLNMCS) (OBJ(APPLNM.OBJ(APPLNMCS)), \
  lo,nomar,noseq,dll,rent,define(_OPEN_SYS,,_DCE_THREADS,MVS), \
  SEARCH(APPLNM.H,'DCEPFX.SEUHD','
   LNGPFX.SCEEH.H',
   'LNGPFX.SCEEH.SYS.H')
```

Figure 21. Compiling in the TSO/E environment

You compile the rest of the C source modules for your DCE applications in the TSO/E environment in similar fashion. Note that the DCE and LE header data sets have an installation-defined high-level qualifier prepended to them.
In Batch: [Figure 22 on page 44] presents sample JCL that you can use to compile your DCE application. All user modifiable entries are italicized in the sample JCL. The cataloged procedure, EDCC, is supplied by the z/OS C/C++ product. Consult the [z/OS C/C++ User's Guide] for more information on using EDCC. Substitute your user ID wherever you see USERPRFX, and your application name wherever you see APPLNM in the example.
Figure 22. Sample JCL to Compile Your DCE Application

There are five major compile steps in Figure 22:
Job Step | Description
---|---
CCCLIENT | Compiles the client portion of your application.
CCCSTUB | Compiles the client stubs generated by the IDL compiler.
CCSERVER | Compiles the server portion of your application.
CCSSTUB | Compiles the server stubs generated by the IDL compiler.
CCSMGRB | Compiles the manager portion of your application server.

Each one of the above job steps can run on its own, provided the job parameters section and USERLIB section specifying the application's header file are completed. The input data set for your application source code is specified by the INFILE parameter. The output data set for your application's object code is specified by OUTFILE parameter. The C/C++ and DCE header files required to compile the DCE applications are specified on the SYSLIB parameter. You may compile more than one client, server, or manager at the same time by adding more job steps as required and modifying the input data sets as appropriate.

**Specifying Compile Options:** The sample JCL contained in Figure 22 on page 44 includes C/C++ compiler options in using the CPARM= statement. Generally, all compile options can be entered at your discretion; however, all DCE applications must be compiled as re-entrant to enable the threading used by the RPC runtime, and, to access APIs in the DCE Dynamic Link Library, the DLL option is recommended. The LONGNAME option should also used.

**Step 9. Link-Editing Your Application**

Once you receive a return code 0 from your compile step, you are ready to link your object code with the DCE RPC runtime library, the C/C++ runtime library, and the TCP/IP runtime library. There are two steps in the linking process: PLKED (prelink step) and LINK (link step).

The C/C++ prelink step is needed because the code is re-entrant and because LONGNAME support is required.

Note to Readers

The prelink step for the example code will end with a return code 4 because the C/C++ runtime entry points, such as printf, strcpy, and so on, are unresolved. You should inspect the prelink output to ensure that none of your application’s entry points are unresolved.

Refer to “Step 8. Compiling the Client and Server Programs” on page 40 for information about how z/OS utilizes the Dynamic Link Library support of the z/OS C/C++ compiler.

**In the Shell:** As in the compile step, use the c89 utility to prelink and link-edit your DCE application object files to create your DCE program which you can run.

**Link-editing Your Client using HFS Files:** Following is an example command to link-edit the object code for the client portion of your DCE application using HFS files:

```
c89 -o applnm_client applnm_client.o applnm_cstub.o -l dce /usr/lib/EUVPDLL.x
```

**Figure 23. Sample Shell Command to Link Your Client using HFS Files**

The above command will create the executable file applnm_client in your current directory.
**Link-editing Your Server using HFS Files:** Following is an example command to link-edit the object code for the server portion of your DCE application using HFS files:

```bash
c89 -o applnm_server applnm_server.o applnm_sstub.o applnm_manager.o -l dce /usr/lib/EUVPDLL.x
```

*Figure 24. Sample Shell Command to Link Your Server using HFS Files*

The above command will create the executable file `applnm_server` in your current directory.

**Link-editing Your Client using PDS:** Following is an example command to link-edit the object code for the client portion of your DCE application using PDS files:

```bash
c89 -o "//@USERPRFX.LOAD(APPLNMCL)" 
  "//@USERPRFX.OBJ(APPLNMCL)" 
  "//@USERPRFX.OBJ(APPLNMCBS)" 
  -l dce /usr/lib/EUVPDLL.x
```

*Figure 25. Sample Shell Command to Link Your Client using PDS*

The above command will create the executable file in `USERPRFX.LOAD(APPLNMCL)`.

**Link-editing Your Server using PDS:** Following is an example command to link-edit the object code for the server portion of your DCE application using PDSs:

```bash
c89 -o "//@USERPRFX.LOAD(APPLNMCL)" 
  "//@USERPRFX.OBJ(APPLNMCL)" 
  "//@USERPRFX.OBJ(APPLNMCBS)" 
  "//@USERPRFX.OBJ(APPLNMSS)" 
  "//@USERPRFX.OBJ(APPLNMRS)" 
  -l dce /usr/lib/EUVPDLL.x
```

*Figure 26. Sample Shell Command to Link Your Server using PDS*

**In TSO/E:** You can link-edit your DCE applications in the TSO/E environment using the CXXMOD EXEC supplied by the z/OS C/C++ program. You may need to customize some of the data sets required by the CXXMOD EXEC for your environment. Refer to the z/OS Language Environment Programming Guide for information about the CXXMOD EXEC.

Following is an example of link-editing the client portion of your DCE application:

```bash
CXXMOD OBJ(APPLNM.OBJ(APPLNMCL),APPLNM.OBJ(APPLNMCBS),'DCEPFX.SEUVEXP(EUVPDLL)')
PLIB('LNGPFX.SCEEOBJ','DCEPFX.SEUVLIB') LIB('LNGPFX.SCEELKED') LOAD(APPLNM.LOAD(APPLNMCL))
```

*Figure 27. Sample TSO/E Command to Link Your Client*

You can link-edit the rest of the server portion of your DCE applications in the TSO/E environment in similar fashion. Note that the above data sets will have different high level qualifiers at your location.
**In Batch:** You can use the EDCPL catalogued PROC, available from the z/OS Language Environment product, to link-edit your client and server. (Shown in Figure 29 on page 48.) Your load module is placed in the data set specified by ‘USERPRFX.APPLNM.LOAD’. The input data set containing your object code is specified in the USERLIB statement. The C/C++ and DCE runtime libraries are specified in the SYSLIB statement. If you create many DCE applications using z/OS DCE, you can customize the EDCPL proc to specify these two runtime libraries for your location.

The sample code in Figure 28 shows the generic JCL to link the client object code and client stub object code with the various runtime libraries to create a load module that you can run.

```
//JOBNAME JOB (ACCOUNT),...your_job_parameters
//**************************************************************************************
/*/ 
/*/ JCL TO LINK THE CLIENT AND CLIENT STUBS OBJECT CODE 
/*/ 
/*/ CUSTOMIZABLE SYMBOLIC PARAMETERS 
/*/ 
/*/ LNGPFX - FOR LE OBJECT LIBRARIES 
/*/ DCEPFX - FOR DCE OBJECT LIBRARIES 
/*/ 
/*/EXEC EDCPL,OUTFILE='USERPRFX.APPLNM.LOAD,DISP=SHR', 
لاعبPEXX='CEE',DCEPFX='DCE'
/*/USERLIB DD DSN=USERPRFX.APPLNM.OBJ,DISP=SHR 
/*/SYSLIB DD DSN=&LNGPFX..SCEEOBJ,DISP=SHR 
// DD DSN=&DCEPFX..SEUVLIB,DISP=SHR 
// DD DSN=&DCEPFX..SEUVEXP,DISP=SHR 
//PLKED.SYSIN DD * 
// INCLUDE USERLIB(APPLNMCS) 
// INCLUDE USERLIB(APPLNMC) 
// INCLUDE SYSLIB(EUVPDL) 
/*/ 
//LKED.SYSIN DD * 
// NAME APPLNMCL(R) 
/*/ 
```

*Figure 28. Sample JCL to Link Your DCE Client*

The sample code in Figure 29 on page 48 shows the generic JCL to link the server code, manager code, and server stub code to create a load module that you can run.
Step 10. Building Your DCE Application

You can combine the compile and link-edit steps to build your DCE applications in the Shell and batch environments.

In the Shell: Your client application is built from the following source files:

- The user-written applnm_client.c client module
- The IDL compiler generated applnm_cstub.c module
- Libraries for the RPC runtime, for IDL stub support, and for the Threads facility. These libraries are included in libdce and the definition side-deck associated with the DCE DLL that you use to link your application.

Your server application is built from the following source files:

- The user-written applnm_server.c client module
- The IDL compiler generated applnm_sstub.c module
- Libraries for the RPC runtime, for IDL stub support, and for the Threads facility. These libraries are included in libdce and the definition side-deck associated with the DCE DLL that you use to link your application.

Use the make facility with a makefile, such as the one presented in Figure 30 on page 49 to build you client and server application programs. This makefile builds your application in one step by running the IDL compiler and the c89 compiler and link-edits your application with the DCE runtime libraries. If you make any changes to one of your DCE application source files, this makefile only builds the application,
that is, the client or server, that is affected by that change. Any intermediate source and object files used to create the executable file are removed. This example uses only HFS files.

IF = applnm

IDL = /bin/idl
IDL_FLAGS = -no_cpp -keep c_source
CFLAGS = -DMVS -D_DCE_THREADS -D_OPEN_SYS -W0,DLL
LIBS = -l dce /usr/lib/EUVPDLL.x

FROMIDL = $(IF).h $(IF)_cstub.c $(IF)_sstub.c
COBJ = $(IF)_client.o $(IF)_cstub.o
SOBJ = $(IF)_server.o $(IF)_sstub.o $(IF)_manager.o

default: $(IF)_client $(IF)_server
 $(IF)_client: $(COBJ)
c89 -o $(if)_client $(COBJ) $(LIBS)
 $(IF)_server: $(SOBJ)
c89 -o $(if)_server $(SOBJ) $(LIBS)

client.o server.o manager.o: $(IF).h
 $(COBJ): $(IF).h
 $(SOBJ): $(IF).h

$(FROMIDL): $(IDL) $(IF).idl
 $(IDL) $(IF).idl $(IDL_FLAGS)

clean:
 rm -f $(FROMIDL) *.o

Figure 30. Sample Makefile to Build Your DCE Application

In the above makefile, the following substitution variables enclosed in parentheses and preceded by $ are used:

IF Your application name designated by applnm
IDL Directory where the IDL compiler resides on your system
IDL_FLAGS IDL compiler options
CFLAGS c89 compiler options
LIBS Name of the DCE runtime library and the definition side-deck associated with the DCE DLL
FROMIDL Output files from the IDL compiler
COBJ The client application object files
SOBJ The server application object files.

You can store this makefile as a text file and assign any name you want to it. Assuming you call your application's makefile, applnm.make, and it is stored in your current working directory, you can build your application by running the make facility as follows:

make -f applnm.make

To remove any intermediate files such as the stubs and object files created during the build, use the following command:

make -f applnm.make clean
Running Your DCE Application

After you have successfully built your DCE application, you can run it by performing the following steps:

1. Starting your server
2. Starting your client
3. Checking your application
4. Stopping your application.
5. Setting your environment

Step 1. Starting Your Server

Once you have successfully linked your application (with a return code 0 for the link step), you can start your DCE application. Before you start your application, ensure that the DCEKERN address space is running on your system by using the Display Active option of the SDSF panel. Always start the server first to avoid the client application from terminating because the server is not available. All DCE applications are POSIX programs by design. Thus, when you run them in TSO/E or batch environments, you need to specify certain runtime options, such as POSIX(ON). These options are not required when you run your application in the Shell.

You must also specify the runtime option STACK(12000) when running your application. The stack storage default per thread is 512K, which severely limits the number of threads.

In the Shell: Figure 31 shows the sample command for running your server application in background mode, which is typically how you would run your server application in the shell. Run your server applications in background mode so that you can gain control and proceed with other tasks while your server is listening.

applnm_server <server_startup_parameters> &

Figure 31. Example Command for Running Your DCE Server in Background Mode

In TSO/E: Usually you do not start DCE server applications in the TSO/E environment because you will not be able to gain control while your server application is running.

For a non-server DCE application, you can start it in TSO/E environment using the TSO/E CALL statement as follows:

CALL 'USERPRFX.LOAD(APPLNM)' 'POSIX(ON),STACK(12000)/ other_parameters' ASIS

In Batch: Figure 32 on page 51 shows the sample JCL for running your server application.
Step 2. Starting Your Client

Once you have established that your server is running correctly, you can start your client program.

In the Shell:  Figure 33 shows the sample command for running your client program.

```
applnm_client <client_startup_parameters>
```

Figure 33. Example Command for Running Your DCE Client

In TSO/E:  You can start up a DCE client application using the TSO/E CALL statement as follows:

```
CALL 'USERPRFX.LOAD(APPLNM)' 'POSIX(ON),STACK(12000)/ other_parameters' ASIS
```

In Batch:  Figure 34 shows the sample JCL for running your client program:

```
//JOBNAME  JOB (ACCOUNT),...your_job_parameters
/*******************
//
//* JCL TO START UP THE CLIENT
//*
/*******************
//STEPNAME  EXEC PGM=CLIENT_NAME,PARM='POSIX(ON)/ <client_startup_parameters>,'
//SYSOUT   DD SYSOUT=*  
//SYSPRINT  DD SYSOUT=*  
//CEEDUMP   DD SYSOUT=*  
//STEPLIB   DD DSN=USERPRFX.APPLNM.LOAD,DISP=SHR
```

Figure 34. Example JCL for Running Your Client
Step 3. Checking Your Application

In The Shell: Once you have started your client and server, you can check their status by entering the process status (ps) command. For example, to find out if your server process is running, enter the following command:

```
ps pid
```

where `pid` is the process ID of your server program job.

For additional information on using the `ps` command, consult `z/OS UNIX System Services User's Guide`.

In TSO/E and Batch: Once you have started your client and server, you can check their status by looking at the SDSF panel in ISPF. On most z/OS systems, you can get to the SDSF panel by entering option ‘S’ on the ISPF command line. Some applications run indefinitely (for example, a server that is listening for a client call), depending on how you design your application. If your job is active, look in the ‘Display active users of the system’ option (select DA in SDSF) to view the output of your application.

If your job is terminated, either because it is programmed to terminate or it stops unexpectedly, look in the ‘Display jobs in the JES2 held output queue’ (option H in SDSF) to view the output.

Step 4. Stopping Your Application

Normal Stopping: The recommended way to stop server applications is to code an external `stop` program to terminate your server applications using DCE APIs like these:

- `rpc_ep_unregister()`
- `rpc_ns_binding_unexport()`
- `rpc_mgmt_stop_server_listening()`

Client programs are normally coded to terminate on their own.

Abnormal Stopping: You can stop your server program from the Shell and TSO/E environments as follows:

From the Shell: To stop your server program or your currently running process, you can send it a quit signal by entering `<CTRL-C>` from your keyboard, or by using the `ps` and `kill` commands.

The `ps` command lists all your processes that are running in the Shell. Note the process identifier (PID) corresponding to your server program. When you find the PID corresponding to the program you want to shut down, issue the `kill` command as follows:

```
kill pid
```

Client programs are normally coded to terminate on their own.

From TSO/E: You can use either the `<PA1>` key, the CANCEL command, or the STOP command to terminate your server application. Note that none of these methods run the TRY/CATCH routines in your application.

You can run the `kill` command from the shell environment, to stop a server application or process running in TSO/E provided you know the PID number.
Step 5. Setting Environment Variables

You can customize the runtime environment for your DCE application by setting certain environment variables. As with OSF DCE, z/OS DCE accepts configuration parameters using environment variables. In many cases, your applications do not need to set any environment variables; instead, they use the defaults set up on your system.

Consider the following environment variables for your DCE applications:

- **LANG**
  
  specifies the locale your program will run in (and therefore the code page it will use). This can either be specified in the shell with the `export` command:

  ```bash
  export LANG=Fr_CA.IBM-037
  ```

  or in the `envar` file:

  ```bash
  LANG=Fr_CA.IBM-037
  ```

- **NLSPATH**
  
  Specifies the directory path z/OS DCE searches for the message catalogs.

  **Note:** The `dce_error_inq_text()` API is influenced by the setting of NLSPATH.

  Normally, this variable is set by the DCE administrator as a default for your system. If not, you should set this variable to `/usr/lib/nls/msg/%L/%N` before you run your z/OS DCE applications. You should also set the `LANG` environment variable to your locale, since its value indirectly affects the value of NLSPATH (see explanation on page 38).

- **RPC_DEFAULT_ENTRY**
  
  Specifies the default entry in the name service database that the RPC NSI import and lookup routines use as a starting point to search, for binding information for a compatible server. An application that uses a default entry name must define this environment variable. The DCE runtime does not provide a default.

- **RPC_DEFAULT_ENTRY_Syntax**
  
  Specifies the syntax of the name provided in the `RPC_DEFAULT_ENTRY` environment variable. In addition, it provides the syntax for those RPC NSI routines that allow a default value for the name syntax parameter. If you do not define `RPC_DEFAULT_ENTRY_Syntax`, the DCE runtime uses the `rpc_c_ns_syntax_dce` name syntax. It can be set to one of the following values:

  0  Use the default value
  1  Unknown (unsupported)
  2  Use DECdns syntax (unsupported)
  3  Use DCE syntax (the default value)
  4  Use ISO OSI X.500 syntax (unsupported)
  5  Use DOD Internet Domain Name Server (unsupported)
  6  Use a UUID string (unsupported)

- **_EUV_SVC_API_DUMPS**
  
  Specifies if a CEEDUMP occurs if invalid parameters are present in any DCE API in your program. It can be one of the following values:

  1  Dumping is enabled
  0  Dumping is disabled.


- **_EUV_RPC_DYNAMIC_POOL**
 Specifies whether a dynamic pool of executor threads is created. It can be one of the following values:

- **Dynamic pool is used (the default on z/OS DCE)**
- **Static pool is used (based on the OSF model).**

**_EUV_EXC_SW_DUMPS**

Specifies if a dump is taken during an exception raised by software. It can be one of the following values:

- **No dump is taken for an exception.**
- **A dump is only taken for an uncaught exception (if no CATCH or CATCH_ALL clause exists). This is the default value.**
- **A dump is taken in all cases except for an explicit catch of an exception (if no CATCH clause exists).**

**_EUV_EXC_ABEND_DUMPS**

Specifies if a dump is taken during an ABEND exception. It can be one of the following values:

- **No dump is taken for an exception.**
- **A dump is only taken for an uncaught exception (if no CATCH or CATCH_ALL clause exists).**
- **A dump is taken in all cases except for an explicit catch of an exception (if no CATCH clause exists).**

**_EUV_RPC_COMM_TIMEOUT**

Used to override the communication timeout default value. The timeout value specifies the relative amount of time a client spends attempting to communicate with a server. The timeout value can be any integer from 0 (zero) to 10, which is the same range of integers accepted by the `rpc_mgmt_set_com_timeout()` API. These integers represent a relative amount of time to spend establishing a client-server relationship. The values are:

- **Attempts to communicate for 1 second.**
- **Attempts to communicate for 2 seconds.**
- **Attempts to communicate for 4 seconds.**
- **Attempts to communicate for 8 seconds.**
- **Attempts to communicate for 15 seconds.**
- **Attempts to communicate for 30 seconds.**
- **Attempts to communicate for 60 seconds.**
- **Attempts to communicate for 120 seconds.**
- **Attempts to communicate for 240 seconds.**
- **Attempts to communicate for 480 seconds.**
- **Attempts to communicate infinitely.**

For further information on the above DCE environment variables, refer to the [z/OS DCE Application Development Reference](https://www.ibm.com/servers/zseries/zos/pdfs/zos_zos90_0273.pdf). You can find information about other DCE environment variables such as `KRBS5CCNAME`, `BIND_PE_SITE`, `TZ` and `_EUV_SVC_MSG_LOGGING` that are normally set as defaults on your system by the DCE Administrator (see [z/OS DCE Administration Guide](https://www.ibm.com/servers/zseries/zos/pdfs/zos_zos90_0273.pdf)). There are also additional environment z/OS DCE variables that are used for serviceability and debugging purposes.
In the Shell: You can set environment variables in the shell two ways using:

- The export command
- The .profile file.

Using the export command: Set an environment variable for your current shell session by running the export command as follows:

```
export var1=x var2=y...
```

For example, to set the `RPC_DEFAULT_ENTRY` to `my_server` you would use the following Shell command:

```
export RPC_DEFAULT_ENTRY=my_server
```

This environment variable is set until the current Shell session ends or until you reset the variable. To verify that the environment variable has been set, use the echo command with the `$` preceding the target environment variable. In the above example, the command would be:

```
echo $RPC_DEFAULT_ENTRY
```

Setting Your .profile file: You can also set your DCE environment variables in your .profile file contained in your home directory. The environment variables in your .profile get set when you enter the Shell session, so you can consider them to be more permanent than the other methods described above. If you change your environment variables in the .profile file, you can issue the .profile command for them to come into effect; otherwise, they are set the next time you enter the Shell. A .profile file contains the export command as follows:

```
export var1=x
export var2=y
export var3=z
```

The file format is a sequential list of environment strings, one per line (or record) unless a string straddles multiple lines because of its length. No blank lines are allowed. As in the UNIX operating system, environment strings must have the following format:

```
VARIABLE=the environment variable definition
```

Variable names are case-sensitive, and no space is allowed between the variable name and the variable definition. If a definition is too long for one line, it can be extended with a backslash (`\`). For example:

```
ANOTHER_VARIABLE=this variable definition will be far too long to \nfit on only one line.
```

All characters preceding the backslash character (including spaces) are considered part of the definition. Trailing spaces after a definition are not included.

Figure 35 on page 56 shows an example envar file. For more information on using DCE environment variables, refer to z/OS DCE Administration Guide.
The `.profile` file is typically used to customize your Shell environment using environment variables.

**In TSO/E:** You can specify environment variables as a run time option on the CALL statement to run your program using the following syntax:

```
ENVAR('var1=x 'var2=y 'var3=z'...)
```

If you want to set the `RPC_DEFAULT_ENTRY` environment variable to `my_server`, you would call your program from a TSO/E command line as follows:

```
CALL 'USERPRFX_LOAD(APPLNM)''POSIX(ON),STACK(12000),ENVAR(''RPC_DEFAULT_ENTRY=my_server'')/other_parameters' ASIS
```

**Using the envar file** An `envar` file typically looks like this:

```
var1=x
var2=y
var3=z
```

If you do not explicitly declare environment variables through the export command (from the command line or through the `.profile` file) in the shell, z/OS DCE looks for them in the default environment variable `envar` file in your home directory. If you want to create this file in a directory other than your home directory, set the value of the `_EUV_ENVAR_FILE` environment variable to a specific path name, as follows:

```
ENVAR(''_EUV_ENVAR_FILE=filename'')
```

**In Batch:** You can specify environment variables using the ENVAR runtime option on the EXEC statement in the JCL that runs your application. You can either specify them individually, using `ENVAR('var1=x, 'var2=y, 'var3=z'...)` syntax, or point to the name of a an `envar` file, as shown below:

```
JOBNAME JOB (ACCOUNT),...your_job_parameters
//*******************************************************
//* JCL TO START UP THE SERVER
//*
//*******************************************************
//STEPNAME EXEC PGM=SERVER_NAME,
// PARM='POSIX(ON),ENVAR(''_EUV_ENVAR_FILE=filename'')/ <options>'
//SYSOUT DD SYSOUT**
//SYSPRINT DD SYSPRINT**
//CEEDUMP DD SYSDUMP**
//STEPLIB DD DSN=USERPRFX.APPLNM_LOAD,DISP=SHR
```

Environment variables set in an `envar` file can be overridden by those specified from the shell using the export command, or by those specified using the ENVAR runtime option from batch or TSO.
Language Environment Runtime Options Considerations

z/OS DCE is supported by the Language Environment runtime environment. When executing a Language Environment application, a number of runtime options can be specified that affect the way your application runs. You can specify these runtime options in a number of ways. See z/OS Language Environment Programming Reference, SA22-7562, for details on the various options and how they are specified.

Caution: Never set the Language Environment runtime option TRAP(OFF) with DCE applications except at the direction of IBM service personnel.

For Application Storage: Some of these runtime options should be used with all z/OS DCE client and server applications since they affect the amount of storage available for the z/OS UNIX and DCE Threads packages. These options are:

LIBSTACK(4K,4K,FREE)
STACK(4K,4K,ANY,FREE)

In addition, if your application uses large numbers of DCE threads (that is, greater than 150), you should also specify:

STORAGE(,,4K)

The above options affect storage use only. If necessary, you should also adjust the HEAP runtime option to meet your application’s needs for heap storage as described below. You should also run your applications with ALL31(ON).

Long running servers may sometimes cause storage exhaustion due to storage fragmentation. If your server tends to use excessive storage and you have ruled out any problems with freeing memory, you may have a fragmentation problem.

To alleviate a fragmentation problem, use a large initial heap allocation. For DCE applications, change the Language Environment default of 64K for initial heap size and heap increments to 1024K and 4K respectively as follows:

HEAP(1024K, 4K, ...)

If your applications run out of storage for threads (that is, you receive a message that pthread_create() failed for storage reasons) and you have implemented the above runtime options, you may need to set the below heap storage as follows:

BELOWHEAP(4096K,256K,ANY,FREE)

For Application Debugging: There are two runtime options that relate to debugging activities:

- TERMTHDACT(DUMP)

  Normally, this runtime option should be set for your application so that in the event of failure, you receive a Language Environment CEEDUMP as well as a traceback. Refer to z/OS Language Environment Debugging Guide GA22-7560. for more information.

- TRACE(ON,64K,NODUMP,LE=1)

  Set this runtime option to obtain a trace table in the Language Environment CEEDUMP. This trace table is referred to as the LE trace. It contains information about runtime library calls to Language Environment, C/C++, sockets, and the z/OS DCE runtime library. You may increase the trace table size by increasing the value of the second parameter, above, or you may request that the trace table
is always dumped when the application ends by specifying DUMP instead of NODUMP, above. Refer to 
"z/OS Language Environment Debugging Guide" for a detailed description of this option.

### Other Runtime Options:
Ensure that the process thread limits are set by setting the MAXTHREADS and MAXTHREADTASKS parameters to a value of 500 or greater. These parameters are contained in the BPXPRMxx parmlib member of the SYS1.PARMLIB data set on your system. The parameters contained in BPXPRMxx control the z/OS UNIX environment, the hierarchical file system, and the sockets file systems. The system uses these parameter values to initialize the kernel.

### Note on Interlanguage Calls
A C/C++ program can communicate with a PL/I program only on the initial thread. Thus, a client program can call a PL/I subroutine on the initial thread as described in "z/OS Language Environment Programming Guide" SA22-7561. Also, a server application can call PL/I and COBOL subroutines before invoking the rpc_server_listen() call in the initial thread.

In the z/OS DCE architecture, the server manager routines always run in created threads. Note that these threads are not what Language Environment refers to as initial threads. Refer to "z/OS Language Environment Programming Guide" for information on the rules, restrictions, and limitations on POSIX conformant C/C++ applications that include z/OS DCE applications.

### Creating a Sample Application: GREET
In this section, you create a simple application using the application development steps introduced in "High-Level Application Development Steps" on page 29. The application steps are repeated with specific annotations enabling you to create the Greet application.

The Greet application is an example of a DCE RPC based application. None of the other DCE services such as Naming, Security, or DTS are used in this application. For simplicity, the server and client programs are located in the same machine.

The client side of the application sends a greeting to the server side of the application. The server prints the client’s greeting and sends a return greeting to the client. The client prints the server’s reply and stops. The server runs indefinitely listening for client RPC calls.

### Fast Path to Running Greet
If you prefer to bypass steps 2 through 7 in the development process and not develop and enter the code, it is provided for you. This includes the necessary IDL and C source code, the Makefile to build the application, and the README for additional information on the application. To get the Greet application up and running quickly:

1. To build and run the Greet application in the Shell, create a directory and copy the Greet application source files to that directory. If you want to build and run Greet in TSO/E or batch, allocate the necessary data sets (as in "1. Creating Files for the Greet Application" on page 59 below), and use the TSO/E OGET command to copy the Greet source files to these application data sets.

2. Proceed to "8. Building the Greet Client and Server Programs" on page 69 and continue with the process to compile, link-edit, and run Greet.

You can find the application source code for the Greet example in the /usr/lpp/dce/examples/greet directory. The source files supplied with z/OS DCE for this example are as follows:

- greet.idl (the IDL file)
If you cannot find the Greet example source files in the above directory, consult your system programmer to find where they are located in your system.

To gain experience using some of the z/OS DCE tools such as the IDL compiler, proceed with the following steps:

1. Creating Files for the Greet Application

**In the Shell:** Create a directory where you want to build and run the Greet application. You do not have to allocate any HFS files in the Shell for the Greet application. The HFS files are created for you when you first edit them, using the TSO/E OEDIT command. To avoid having to key in the source code, you can copy the Greet source files from the `/usr/lpp/dce/examples/greet` directory to your directory.

**In ISPF::** To allocate the required data sets for Greet, enter:

```
IDLALLOC
```

A menu is presented by IDLALLOC. In the `DSN PREFIX` area, fill in `USERPFX.GREET`. For more information, see page 33 and [Figure 2 on page 34](#).

IDLALLOC will create the following partitioned data sets automatically.

- `USERPRFX.GREET.ACF`
- `USERPRFX.GREET.C`
- `USERPRFX.GREET.H`
- `USERPRFX.GREET.IDL`
- `USERPRFX.GREET.OBJ`

After you have allocated the necessary data sets, copy the source files to the appropriate data set using the TSO/E OGET command. The Greet application does not require attribute configuration parameters and thus the ACF data set is not required. You can list these data sets to verify they have been created by using the `Data Set List Utility` in ISPF (option 3.4).

2. Generating a UUID and an IDL File

Next, you must create a UUID to uniquely label the Greet interface, and create a template for the Greet application IDL file.

**In the Shell:** Use the example command in Figure 37 to generate a UUID and create a template for the Greet IDL file:

```
uuidgen -i -o greet.idl
```

*Figure 37. Shell Command to run the UUID Generator for Greet*

In the above example, the `uuidgen` command is run with the `-i` option, which specifies that a skeletal interface definition file is to be generated that includes the UUID. The `-o` option redirects the output from
your screen to the `greet.idl` file in your current directory. For a complete description of the input parameters for the UUID generator, see [z/OS DCE Application Development Reference](#).

**In TSO/E:** You can also generate a UUID in foreground mode by entering the following TSO/E command:

```
uuidgen -i -o greet "/USERPRFX.GREET.IDL"
```

**In Batch:** Use the example JCL contained in Figure 38 to generate a UUID for the Greet application. In addition to generating a UUID for your application, running this JCL creates a template for your application’s IDL file.

```
//JOBNAME JOB (ACCOUNT)...your_job_parameters
//**********************************************************************************************
//* Execute UUIDGEN using JCL                                *
//**********************************************************************************************
//UUIDGEN EXEC UUIDGEN,                                      //
//   PARMS="-i -o "/GREET.IDL(GREET)"
```

*Figure 38. Generating a UUID for the Greet Application in Batch*

The contents of the Greet IDL file generated in any of the above environments is as shown in Figure 39.

```
[
   uuid(20818313-4143-19ea-a6e0-0000dce12345),
   version(1.0)
]
interface INTERFACENAME
{
}
```

*Figure 39. Example of the Greet Application IDL File*

**Note:** To avoid any potential conflicts with other instances of the same Greet example and ensure that you are running a unique instance, you should generate your own interfaceUUID using the UUID Generator and replace the interface UUID in the IBM-supplied Greet IDL file with it.

### 3. Naming the Greet Interface

You can edit the Greet IDL file member using your favorite MVS/ESA™ or ISPF editor. Replace the string `INTERFACENAME` with `greet`, the interface name for Greet. The convention is to name the interface the same as the IDL file. The content of Greet IDL file should be similar to the following:

```
[
   uuid(20818313-4143-19ea-a6e0-0000dce12345),
   version(1.0)
]
interface greet
{
}
```

*Figure 40. Naming the Greet Interface*
4. Defining the Interface Operations

Within the braces, define the operation comprising the Greet interface. In this example, there is only one operation called greet_rpc. Figure 41 shows the contents of the Greet IDL file.

```
[uuid(20818313-4143-19ea-a6e0-0000dce12345),
version(1.0)]

interface greet {
    const short int STR_SZ=128;
    void greet_rpc (  
        [in] handle_t h,  
        [in] char client_greeting[STR_SZ],  
        [out] char server_reply[STR_SZ]
    );
}
```

Figure 41. The Greet Application Interface Operations

The GREET interface operations are defined as follows:

1. The first line defines the name of the operation, greet_rpc, and indicates by the void declaration that it has no meaningful return value.
2. In the next line, handle_t h is a declaration defining h as a primitive binding handle. It is meaningful to the RPC runtime library.
3. The next two lines specify the client_greeting and server_reply arguments to the operation. The first is a string that is passed from the client to the server — denoted by “[in]”; the second is a string returned from the server back to the client — denoted by “[out].”

5. Compiling the Greet Interface with the IDL Compiler

After you have written the Greet interface definition, you can use one of the following methods to compile it. In the following examples, you run the IDL compiler with the verbose option (-v), which means that the IDL compiler outputs informational messages when compiling the IDL file. The -no_cpp directive specifies that the C preprocessor is not run. In addition, the -keep c_source option in the examples retains the output files (the stubs) as C source modules. The default for the IDL compiler, if the -keep c_source option is not specified, is to invoke the compiler and retain the object modules only, for the output files.

The IDL compiler can also produce stubs in object format by invoking the C/C++ compiler. The .c portion of the output file is replaced by .o.

The IDL compiler offers many options that enable you to choose the C/C++ compiler or preprocessor commands that are invoked, which directories are searched for imported files, which of the possible output files are generated, and how the output files are named. When you compile the definition of a remote interface, you must ensure that the system IDL directory is among those that the IDL compiler searches when it searches for imported files, because any remote interface implicitly imports nbase.idl.
**In the Shell:** Use the sample command shown in Figure 42 to compile the Greet IDL file, producing the header and stub files:

```shell
idl greet.idl -no_cpp -v -keep c_source
```

*Figure 42. Sample Command to Invoke the IDL Compiler in the Shell*

The IDL compiler creates three output files for Greet the Greet application. These files are compiled along with the Greet application source code. The files are:

- `greet.h` (The Greet header file)
- `greet_cstub.c` (The Greet client stub)
- `greet_sstub.c` (The Greet server stub)

The Greet header file contains the definitions and declarations derived from the input `greet.idl` file that are for general use in the development source code. Its contents are shown in *Figure 45 on page 63*.

**In TSO/E:** Use the sample CLIST command, IDL, shown in Figure 43, to compile the Greet IDL file and produce the files listed above. Note that IDL handles PDS only.

```shell
idl GREET -no_cpp -v -keep c_source -userpfx USERPRFX
```

*Figure 43. Sample TSO/E Command to Run the IDL Compiler*

**In Batch:** In batch, use the sample JCL presented in Figure 44:

```jcl
//JOBNAME JOB (ACCOUNT)...your_job_parameters
//******************************************************************************
//* RUN THE IDL Compiler    *
//******************************************************************************
//IDLCOMP EXEC IDL,USERPFX='USERPRFX.GREET',
 // PARM='greet -no_cpp -v -keep c_source'
```

*Figure 44. Sample JCL to Run the IDL Compiler*

For the TSO/E and batch environments, the IDL compiler creates three files with associated members for Greet. These data sets are compiled along with the Greet application source code. The data sets are:

- `header` *USERPRFX.H(GREET)*
- `client stub` *USERPRFX.C(GREETCS)*
- `server stub` *USERPRFX.C(GREETSS)*

*USERPRFX.H(GREET)* contains the definitions and declarations derived from the input *USERPRFX.IDL(GREET)* data set that are for general use in the development source code. Its contents are shown in *Figure 45 on page 63*. 
/* Generated by IDL compiler version OSF DCE T1.1.0-03 */
#ifndef greet_v1_0_included
#define greet_v1_0_included
#ifndef IDLBASE_H
#include <dce/idlbase.h>
#endif
#include <dce/dcerpcmsg.h>
#include <dce/rpc.h>
#ifdef __cplusplus
extern "C" {
#endif
#ifndef nbase_vgz'ro@ot_gz'ro@ot_included
#include "dce/nbase.h"
#endif
#define STR_SZ (128)
extern void greet_rpc(
#ifdef IDL_PROTOTYPES
/* [in] */ handle_t h,
/* [in] */ idl_char client_greeting[128],
/* [out] */ idl_char server_reply[128]
#endif
);
typedef struct greet_v1_0_epv_t {
void (*greet_rpc)({
#ifdef IDL_PROTOTYPES
/* [in] */ handle_t h,
/* [in] */ idl_char client_greeting[128],
/* [out] */ idl_char server_reply[128]
#endif
});
} greet_v1_0_epv_t;
extern rpc_if_handle_t greet_v1_gz'ro@ot_c_ifspec;
extern rpc_if_handle_t greet_v1_gz'ro@ot_s_ifspec;
#ifdef __cplusplus
}
#endif
#endif

Figure 45. The Greet Application Header File

The stubs, one for your server and one for your client, contain the runtime code for marshalling and unmarshalling data, message handling, and other details of network communications management.

You can browse the Greet client and server stub files to see their contents. They are not listed here because they are quite lengthy. You do not make any changes to the stub code.

6. Writing the Greet Server and Manager Code

In the Shell, the C code you write for the Greet server is contained in the HFS file greet_server.c. The Greet manager, which forms part of the server side of the Greet application, is contained in greet_manager.c.

In the TSO/E or batch environments, the C code you write for the Greet server is contained in the USERPRFX.GREET.C data set with the following members:

- MANAGER
The Greet manager, which forms part of the server side of the Greet application, contains the implementation of the greet_rpc operation. It contains the C source code shown in Figure 46:

```c
#include <stdio.h>
#include "greet.h"

void greet_rpc(handle_t h,
               char *client_greeting,
               char *server_reply)
{
    printf("The client says: %s\n", client_greeting);
    fflush(stdout);
    strncpy(server_reply, "Hi client!", STR_SZ);
}
```

Figure 46. Greet Manager Source Code

The Greet server contains the main portion of the Greet application. It registers its interface with the RPC runtime, and then listens for service requests from Greet clients. The server goes through the following steps. The numbers shown in blocks refer to Figure 47 on page 65.

1. The server includes all the necessary header files. Referring to Figure 47 on page 65, notice the file `<dce/dce_error.h>` has been included. This header file contains a defined constant, `dce_c_error_string_len`, a typedef, `dce_error_string_t` and a prototype of the function `dce_error_inq_text()`. Notice also that the `<dce/exc_handling.h>` file has been included. It is required so the server can handle any unexpected interrupts while it is listening for client requests. With this exception handling, the server application can be terminated gracefully. That is, actions such as unregistering its interface from the endpoint map can be taken prior to terminating the server application.

In the main routine, the server:

2. Makes all the necessary variable declarations.

3. Sets the locale for this application to the default locale for your system.

4. Checks to ensure that only one argument is received at start up, and prints an error message if more or fewer arguments are received.

5. The server then calls the `rpc_server_use_all_protseqs()` function to register all protocol sequences that are supported by the operating system and the DCE runtime, with the DCE runtime. Also, with this function call, the server obtains an endpoint for each supported protocol from the runtime on which it listens for client requests.

6. The server calls the `rpc_server_register_if()` function to register its interface with the RPC runtime by supplying its interface specification.

7. The server calls the `rpc_server_inq_bindings()` function to obtain a vector of binding handles that can be used to register the server’s endpoint. The server then obtains, prints, and frees a string binding. The binding is printed in this example so that later you can provide it as a parameter when you start your client. This enables your client to find the server (through this binding). This is done only to simplify the example; normally, a binding would be obtained from the name service.

8. To begin listening for RPC requests, the server calls `rpc_server_listen()` function. This call is placed within the TRY of a TRY, CATCH_ALL, ENDTRY sequence, so that if the server receives an exception while it is listening, it prints out a status message before it exits. For more information on TRY and CATCH sequences, see Chapter 4, “Threads” on page 149.
The source code for the Greet server is shown in Figure 47 on page 65. To obtain more information on the syntax and use of the RPC API calls presented in this example, refer to the appropriate section of Z/OS DCE Application Development Reference for the particular API call.

```c
#pragma runopts(stack(12K,4K,ANY,KEEP))
#include <stdio.h>
#include <locale.h>
#include <dce/dce_error.h>
#include <dce/exc_handling.h>
#include "greet.h"

#define MAX_CONCURRENT_CALLS 5

/* In the first part of the main function, the server calls
rpc_server_use_all_protseqs to use all protocol
sequences that are supported on its host both by the
runtime library and the operating system. */

int main (int argc, char *argv[]) {
    rpc_binding_vector_p_t bvec;
    unsigned long st;
    int error_inq_st;
    dce_error_string_t error_text;
    idl_boolean validfamily;
    idl_char string_binding;
    int i;
    FILE *bindingfile;

    setlocale(LC_ALL, "");

    if (argc != 1) {
        fprintf (stderr, "Usage: %s\n",argv[0]);
        fflush (stderr);
        exit(1);
    }

    /* Calling rpc_server_use_all_protseqs to obtain an endpoint
for each protocol sequence supported by the RPC runtime and
the operating system */

    rpc_server_use_all_protseqs(MAX_CONCURRENT_CALLS, &st);
    if ([st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot register protocol seqs - %s\n", error_text);
        fflush(stderr);
        exit(1);
    }

    /* Calling rpc_server_register_if to register its interface with
the RPC runtime by supplying its interface specifier */

    rpc_server_register_if(greet_v1_0_s_ifspec, NULL, NULL, &st);
    if ([st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot register interface - %s\n", error_text);
        fflush(stderr);
        exit(1);
    }
}
```

Figure 47 (Part 1 of 2). Greet Server Source Code
/* Calling rpc_server_inq_bindings to obtain a vector of binding handles that can be used to register the server's endpoint. The server then obtains, prints, and frees a string binding */

rpc_server_inq_bindings(&bvec, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    printf(stderr, "Cannot inquire bindings - %s\n", &error_inq_st);
    fflush(stderr);
    exit(1);
}

printf("Bindings:\n");
bindingfile = fopen("dd:binding","wb+, type=record");
fflush(stdout);

for (i = 0; i < bvec->count; i++) {
    rpc_binding_to_string_binding(bvec->binding_h[i], &string_binding, &st);
    printf("%s\n", (char *)string_binding);
    fwrite((char *)string_binding, strlen((char *)string_binding), 1, bindingfile);
    fflush(stdout);
    rpc_string_free(&string_binding, &st);
    fclose(bindingfile);
}

/* To begin listening for RPC requests, the server calls rpc_server_listen. This call is placed within the TRY of a TRY, CATCH_ALL, ENDTRY sequence, so that if the server receives a signal while it is listening, the CATCH_ALL code will allow the server to shut down gracefully. */

TRY {
    printf("Listening...\n");
    fflush(stdout);
    rpc_server_listen(MAX_CONCURRENT_CALLS, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        printf(stderr, "Error: %s\n", error_text);
        fflush(stderr);
    }
}

CATCH_ALL {
    printf("Server GREET shutting down\n");
    fflush(stdout);
}

ENDTRY;
return(0);

Figure 47 (Part 2 of 2). Greet Server Source Code

7. Writing the Greet Client Code

In the Greet client program, you provide the Greet server binding information as start up parameters. The Greet server prints out this information as part of its initialization procedure. The Greet client uses this binding information to establish communications with the Greet server. The client makes the RPC function call, passing along the required input parameter, that is, the greeting, to the server. It then prints out the server's reply. The specific steps follow. The numbers shown in blocks refer to Figure 48 on page 68

1 The client includes the necessary header files.
In the main routine, the client:

2. Makes all the necessary variable declarations.
3. Sets the locale for this application to the default locale for your system.
4. Checks to ensure that five input arguments are received at start up, and prints an error message if an incorrect number of parameters are received.
5. Assigns input arguments to the variables representing the constituent parts of the binding to the server. These are:
   1. Protocol sequence (protseq)
   2. Host internet identifier (hostid)
   3. Endpoint on the server is listening (endpoint).

The client then calls the `rpc_string_binding_compose()` function to derive a string binding from the above binding components.

6. Calls the `rpc_binding_from_string_binding()` function to obtain an RPC binding. The client prints out the string binding for checking purposes.

7. Assigns variable MAX_PASS the value of the last input argument received. It enters a loop where it makes the `greet_rpc` function call and prints out the server's reply the number of times specified by MAX_PASS.

The source code for the Greet client is shown in Figure 48 on page 68. To obtain more information on the syntax and use of the RPC functions presented in this example, refer to the appropriate section of z/OS DCE Application Development Reference for the particular function.
```c
#include <stdio.h>
#include <locale.h>
#include <dce/dce_error.h>
#include "greet.h"

int main(int argc, char* argv[]) {
    handle_t h;
    unsigned long st;
    int error_inq_st;
    dce_error_string_t error_text;
    idl_char string_binding, protseq, hostid, endpoint;
    static idl_char nil_string[] = "";
    int i, MAX_PASS;
    char reply[STR_SZ];

    setlocale(LC_ALL, "");
    if (argc != 5) {
        fprintf(stderr, "Usage: %s protseq hostid endpoint passes\n", argv[0]);
        fflush(stderr);
        exit (1);
    }

    protseq = (idl_char *) argv[1];
    hostid = (idl_char *) argv[2];
    endpoint = (idl_char *) argv[3];

    rpc_string_binding_compose(nil_string, protseq, hostid, endpoint, nil_string, &string_binding, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Can't compose string binding - \s\n", error_text);
        fflush(stderr);
        exit(1);
    }

    rpc_binding_from_string_binding(string_binding, &h, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot get binding from string binding %s - %s\n", string_binding, error_text);
        fflush(stderr);
        exit(1);
    }

    printf("Bound to %s\n",string_binding);
    fflush (stdout);

    MAX_PASS= atoi(argv[4]);
    for (i=1; i <= MAX_PASS; i++) {
        greet_rpc(h, "Hello Server!", reply);
        printf("The Greet Server said: \s\n", reply);
        fflush(stdout);
    }

    return(0);
}

Figure 48. Greet Client Source Code
```
8. Building the Greet Client and Server Programs

Once you have completed writing your client, server, and manager source code, you build this code along with the client and server stubs using the C/C++ compiler.

**In the Shell:** Use the `make` facility with a makefile, such as the one presented in Figure 49 to build your Greet client and server application programs. This makefile builds the Greet application in one step by running the IDL compiler and the `c89` compiler, and link-edits your application with the DCE runtime libraries. Any intermediate source and object files used to create the Greet executable file are removed. This example uses only HFS files.

```bash
IF = greet
IDL = /bin/idl
IDL_FLAGS = -no_cpp -v -keep c_source
CFLAGS = -DMVS -D_DCE_THREADS -D_OPEN_SYS -D_OPEN_THREADS -Wgz'ro@ot,DLL
LIBS = -l dce /usr/lib/EUVPDLL.x
FROMIDL = $(IF).h $(IF)_cstub.c $(IF)_sstub.c
COBJ = $(IF)_client.o $(IF)_cstub.o
SOBJ = $(IF)_server.o $(IF)_sstub.o $(IF)_manager.o

default: $(IF)_client $(IF)_server
$(IF)_client: $(COBJ)
c89 -o $(IF)_client $(COBJ) $(LIBS)
$(IF)_server: $(SOBJ)
c89 -o $(IF)_server $(SOBJ) $(LIBS)

$(COBJ): $(IF).h
$(SOBJ): $(IF).h

$(FROMIDL): $(IDL) $(IF).idl
$(IDL) $(IF).idl $(IDL_FLAGS)

clean:
  rm -f $(FROMIDL) *.o
```

*Figure 49. Makefile to Build the Greet Application*

**In Batch:** [Figure 50 on page 70](#) presents the JCL to compile the Greet source code. The user ID `USERPRFX` is used in this example. You must substitute your own user ID in place of `USERPRFX` before compiling your application.
//JOBNAME JOB (ACCOUNT),...your_job_parameters

//*******************************************************************************************
//* JCL TO COMPILE THE GREET CLIENT, SERVER, AND STUBS CODE
//*
//*******************************************************************************************
// CUSTOMIZABLE SYMBOLIC PARAMETERS

//*******************************************************************************************
// DCEPFX - FOR DCE HEADER FILES
// LNGPFX - FOR LE HEADER FILES
//*******************************************************************************************

//*******************************************************************************************
// CCCLIENT EXEC EDCC,INFILE='USERPRFX.C(GREETCL)',
// CPARM='LO,NOMAR,NOSEQ,DLL,RENT,DEFINE(_OPEN_SYS,_DCE_THREADS,MVS)',
// OUTFILE='USERPRFX.OBJ(GREETCL),DISP=SHR',
// DCEPFX='DCE',LNGPFX='CEE'
// USERLIB DD DSN=USERPRFX.GREET.H,DISP=SHR
// SYSLIB DD DSN=DCEPFX...SEUVHDR,DISP=SHR
// DD DSN=LNGPFX...SCEEH.H,DISP=SHR
// DD DSN=LNGPFX...SCEEH.SYS.H,DISP=SHR

//*******************************************************************************************
// CCMSTUB EXEC EDCC,INFILE='USERPRFX.C(GREETCS)',
// CPARM='LO,NOMAR,NOSEQ,DLL,RENT,DEFINE(_OPEN_SYS,_DCE_THREADS,MVS)',
// OUTFILE='USERPRFX.OBJ(GREETCS),DISP=SHR',
// DCEPFX='DCE',LNGPFX='CEE'
// USERLIB DD DSN=USERPRFX.GREET.H,DISP=SHR
// SYSLIB DD DSN=DCEPFX...SEUVHDR,DISP=SHR
// DD DSN=LNGPFX...SCEEH.H,DISP=SHR
// DD DSN=LNGPFX...SCEEH.SYS.H,DISP=SHR

//*******************************************************************************************
// CCMSGRB EXEC EDCC,INFILE='USERPRFX.C(GREETMR)',
// CPARM='LO,NOMAR,NOSEQ,DLL,RENT,DEFINE(_OPEN_SYS,_DCE_THREADS,MVS)',
// OUTFILE='USERPRFX.OBJ(GREETMR),DISP=SHR',
// DCEPFX='DCE',LNGPFX='CEE'
// USERLIB DD DSN=USERPRFX.GREET.H,DISP=SHR
// SYSLIB DD DSN=DCEPFX...SEUVHDR,DISP=SHR
// DD DSN=LNGPFX...SCEEH.H,DISP=SHR
// DD DSN=LNGPFX...SCEEH.SYS.H,DISP=SHR

//*******************************************************************************************
// CCSERVER EXEC EDCC,INFILE='USERPRFX.C(GREETSR)',
// CPARM='LO,NOMAR,NOSEQ,DLL,RENT,DEFINE(_OPEN_SYS,_DCE_THREADS,MVS)',
// OUTFILE='USERPRFX.OBJ(GREETSR),DISP=SHR',
// DCEPFX='DCE',LNGPFX='CEE'
// USERLIB DD DSN=USERPRFX.GREET.H,DISP=SHR
// SYSLIB DD DSN=DCEPFX...SEUVHDR,DISP=SHR
// DD DSN=LNGPFX...SCEEH.H,DISP=SHR
// DD DSN=LNGPFX...SCEEH.SYS.H,DISP=SHR

Figure 50. Sample JCL to Compile Greet
Once you receive a return code 0 from your compile step, you are ready to link the Greet object code with the DCE RPC runtime library, the definition side-deck associated with the DCE DLL, the C/C++ runtime library, and the TCP/IP runtime library.

The sample code in Figure 51 shows the JCL to link-edit the Greet server code, manager code, and server stub code to create a load module that you can run.

```plaintext
//JOBNAME JOB (ACCOUNT),...your_job_parameters
//*******************************************************************************
// * JCL TO LINK THE GREET SERVER, MANAGER AND SERVER STUBS OBJECT CODE
// *
//*******************************************************************************
// CUSTOMIZABLE SYMBOLIC PARAMETERS
//*******************************************************************************
// * LNGPFX - FOR LE OBJECT LIBRARIES
// * DCEPFX - FOR DCE OBJECT LIBRARIES
// *
//*******************************************************************************
//LKSERVER EXEC EDCPL,OUTFILE='USERPRFX.GREET.LOAD,DISP=SHR',
// LNGPFX='CEE',DCEPFX='DCE'
//USERLIB DD DSN=USERPRFX.GREET.OBJ,DISP=SHR
//SYSLIB DD DSN=LNGPFX..SCEEOBJ,DISP=SHR
// DD DSN=DCEPFX..SEUVLIB,DISP=SHR
// DD DSN=DCEPFX..SEUVEXP,DISP=SHR
//PLKED.SYSIN DD *
INCLUDE USERLIB(GREETSR)
INCLUDE USERLIB(GREETMR)
INCLUDE USERLIB(GREETSS)
INCLUED SYSLIB(EUVPDLL)
/
//LKED.SYSIN DD *
NAME GREETSR(R)
/

Figure 51. Sample JCL to Link the Greet Server Code
```

The sample code in Figure 52 on page 72 shows the JCL to link-edit the Greet client object code and client stub object code with the various runtime libraries to create a load module that you can run.
9. Starting the Greet Server

Once you have successfully linked the Greet application (a return code 4 on the prelink step and return code 0 for the link step), you can begin to run it.

Ensure that the DCEKERN address space is running on your machine before you start the Greet application. DCEKERN contains the DCE daemons that are configured to run on your machine. Check if it is running using the Display Active option of the SDSF panel.

Start the Greet server first to avoid the client application terminating because of server unavailability.

**In the Shell:**  Figure 53 shows the sample command for running the Greet server application in background mode:

```
greet_server &
```

**Figure 53. Running the Greet Server in Background Mode**

Within a short time, the Greet server is registered and in listening mode. You should see output similar to that in Figure 54:

```
Bindings:
ncadg_ip_udp:9.21.6.97[1036]
nacn_ip_tcp:9.21.6.97[1668]
Listening...
```

**Figure 54. Greet Server Output in Listening Mode**
**In Batch:** Figure 55 shows the sample JCL for running the Greet server application:

```
//JOBNAME JOB (ACCOUNT),...your_job_parameters
//******************************************************************************
// JCL TO STARTUP THE GREET SERVER
//******************************************************************************
//GREETSR EXEC PGM=GREETSR,PARM='POSIX(ON)/'
//BINDING DD DSN=USERPRFX.GREET.LOAD,DISP=SHR
//SYSOUT DD SYSOUT=g+O#SJ
//SYSPRINT DD SYSOUT=g+O#SJ
//SYSUDUMP DD SYSOUT=g+O#SJ
//STEPLIB DD DSN=USERPRFX.GREET.LOAD,DISP=SHR
```

*Figure 55. Example JCL for Running the Greet Server*

In this example, the program name GREETSR is passed to the Greet server in argv[0].

After you submit this JOB to start the Greet server, check its status using the SDSF panel. You will initially see output similar to that in Figure 56. The binding you obtain will be different from this example. You will need to pass the components of this binding as arguments to your client when you start it up.

**Bindings:**

```
ncadg_ip_udp:9.21.6.97[1036]
```

*Figure 56. Initial Output of Greet Server*

**Note:** The Greet application is designed to run optimally in an interactive environment such as the z/OS UNIX System Services shell, as you need to see the bindings that are printed out from the Greet server and pass them in as input parameters to the Greet client application. In the batch environment, the `printf()` output from the server may not be visible if the server is in a quiescent state. If this is the case, you may have to replace the `printf()` with an `fopen()` to write to a file or data set, and browse it for the binding information. In the batch environment, you may find it more convenient to run applications that do not depend that you note the output from the server, that is, examples that use CDS to find the location of the server. Except for Greet and Greet1, all the DCE RPC example applications use CDS, and are more suitable to be run in batch.

## 10. Starting the Greet Client

Once you have established that the Greet server is running correctly, start the Greet client.

In this example, four arguments are passed to the Greet client program, as follows:

- The protocol sequence — `ncadg_ip_udp` (protocol sequence `ncacn_ip_tcp` could also have been chosen)
- The Greet server's host internet address — 9.21.6.97
- The Greet server's end point — 1036 (If `ncacn_ip_tcp` is used, then end point 1668 would be used here)
- The number of times to invoke the RPC call to the Greet server — 3.

The first three arguments passed in the GPARM statement make up a complete string binding: `ncadg_ip_udp:9.21.6.97[1036]`. 
**In the Shell:** Figure 57 shows the sample command for running your client program:

```bash
$ greet_client ncadg_ip_udp 9.21.6.97 1036 3
```

*Figure 57. Example Command for Running the Greet Client in the Shell*

**In Batch:** Figure 58 shows the sample JCL for running your client application:

```jcl
//JOBNAME JOB (ACCOUNT),...your_job_parameters
//*****************************************************************************
//
// JCL TO STARTUP THE GREET CLIENT
//
//*****************************************************************************
//GREETCL EXEC PGM=GREETCL,PARM='POSIX(ON)/ ncadg_ip_udp 9.21.6.97 1036 3'
//SYSOUT DD SYSOUT=g+O#SJ
//SYSPRINT DD SYSOUT=g+O#SJ
//SYSUDUMP DD SYSOUT=g+O#SJ
//STEPLIB DD DSN(USERPRFX.GREET.LOAD,DISP=SHR)
```

*Figure 58. Sample JCL for Running the Greet Client*

After you submit this JOB to start the Greet client, check its status using the SDSF Held Output Display panel. After waiting for your client to finish running, check its output. It should be similar to that in Figure 59. Of course, your binding will be different from this example.

Bound to ncadg_ip_udp:9.21.6.97[1036]
The Greet Server said: Hi client!
The Greet Server said: Hi client!
The Greet Server said: Hi client!

*Figure 59. Greet Client Output*

Checking the Greet server output once more in the SDSF Display Active panel, you see output similar to that in Figure 60:

```plaintext
Bindings:
ncadg_ip_udp:9.21.6.97[1036]
ncacn_ip_tcp:9.21.6.97[1668]
Listening...
The client says: Hello Server!
The client says: Hello Server!
The client says: Hello Server!
```

*Figure 60. Final Output for the Greet Server*

The Greet client program terminates on its own. To terminate the Greet server program, refer to "Step 4, Stopping Your Application" on page 52.
Chapter 2. Extending the Greet Application

In this chapter, you add a few variations to the Greet application created in Creating a Sample Application: GREET on page 58. These variations show how some of the DCE services available enhance your distributed application. You are shown the following:

1. Logging into DCE as a DCE principal, passing your DCE permissions to your client or server application, and running your DCE applications as a different DCE principal.

2. Different methods for your client application to search and locate its compatible server using the DCE Host Daemon (DCE Host Daemon) and the RPC Name Service Interface (NSI). Note that you are shown how to authenticate your application to the DCE Security daemon in order to write to the DCE Host Daemon and the CDS daemon.


4. An application server with multiple objects and types. In this case, the Greet server application is extended to respond in one of three different languages, depending on the nationality specified in the client routine.

Logging Into DCE

All the example applications in this chapter must run in authenticated mode as access to the DCE Host Daemon or to the CDS data base is controlled by DCE Security. This requires that you log into DCE Security as a DCE user principal and pass the permissions of that principal to the DCE application you run. For example, if you log into DCE Security as the principal Ricardo, and this principal possesses certain permissions such as write access to the CDS namespace, you can pass those permissions to your DCE application. That is, your application can inherit your user context and thus possess the same permissions as you.

Prior to logging into DCE Security, consult your DCE administrator, or see z/OS DCE Administration Guide to set up your user principal name, account, password, group, and all necessary permissions. Once your account is set up, you can log into DCE from either the Shell, TSO/E, or batch environments.

In the Shell, enter the following command:

dce_login <principal_name> <password>

In TSO/E, enter the following command:

DCELOGIN <principal_name> <password>

In batch, use the following JCL to run the DCELOGIN catalogued procedure:

```plaintext
//JOBNAME JOB (ACCOUNT)
//*******************************************************************************
//*
//* JCL TO STARTUP A DCE APPLICATION
//*
*******************************************************************************
//STEP1 EXEC DCELOGIN,PARM='principal_name password'
//STEP2 EXEC PGM=your_program,PARM='POSIX(ON)'
//STEPLIB DD DSN=USERPRFX.<APPLNM>.LOAD,DISP=SHR
```

Figure 61. Logging into DCE in Batch

© Copyright IBM Corp. 1994, 2001 75
In the Shell or TSO/E environments, you are prompted for your principal name and password if you do not enter them.

Each time you log into DCE using any of the above methods, a credentials cache file is created that corresponds to the account or principal name from which you log in. This file contains your tickets that validates your identity to DCE Security. By default, an HFS file called krb5ccname, is created in your home directory that contains the KRB5CCNAME environment variable. This environment variable points to your credentials cache file created by the dcelogin mechanism.

The contents of the krb5ccname file are similar to the following:

"KRB5CCNAME=FILE:file-name"

The credentials cache file designated by file-name is stored in the /opt/dcelocal/var/security/creds directory.

Note that the file containing the _EUV_KRB5CCNAME_FILE environment variable can also be a sequential data set which you allocate as follows:

ALLOC FI(EUVSKRB5) DS('USERPRFX.KRB5CCNAM.FILE')SHR REUSE

This data set can be of any size with logical record length greater than 80. It only contains a single line. This data set is only used by DCE applications running in TSO/E and batch environments.

You can also specify an HFS file by using PATH= on your ALLOC command as follows:

ALLOC FI(EUVSKRB5) PATH'<HFS-file-name>' PATHOPTS(ORDWR) PATHDISP(KEEP,KEEP)

### Changing Your Login Context

You may need to run some of your batch DCE applications under a different identity. If you have multiple identities in DCE, you can login multiple times to DCE and switch identities. In DCE, you switch identities by inheriting a login context which is your identity and associated privileges that are specified by your credentials cache file. For example, you may want to run your client program as a different principal than your server program, with different DCE permissions. Contrast this with the MVS security paradigm that you can log on only as one identity.

**In the Shell:** Use the export command to specify the HFS credentials cache file that you want your application to inherit. For example, if you are DCE logged-in as principal A and wish to DCE log in as principal B while preserving the credentials for principal A, enter the following before DCE logging in as principal B:

export _EUV_SEC_KRB5CCNAME_FILE='HOME/krb5ccname1'

When you subsequently run your DCE application, it will inherit the credentials based on the setting of the _EUV_SEC_KRB5CCNAME_FILE environmental variable. If it points to the krb5ccname1 file your application will inherit the credentials of principal B; if it points to krb5ccname it will inherit the credentials of principal A.
In TSO/E:  To set your login context in TSO/E, allocate the EUVSKRB5 data set as follows, prior to logging into DCE:

ALLOC FI(EUVSKRB5) DS('USERPRFX.TICKET1')SHR REUSE

To change your context in TSO/E, allocate another data set of a different name, prior to logging into DCE as follows:

ALLOC FI(EUVSKRB5) DS('USERPRFX.TICKET2')SHR REUSE

You can log in as different identities in each case. The data sets you allocate each point to a different ticket cache thereby preserving your different identities.

In Batch:  To switch your login context in batch, specify a different EUVSKRB5 DD name in your JCL as in Figure 62. Note that the data set referenced by EUVSKRB5 must be allocated as described above.

//JOBNAME JOB (ACCOUNT)

Figure 62. Changing Your Login Context for Batch Applications

To specify an HFS file as the EUVSKRB5 file, specify the following in the above JCL:

//EUVSKRB5 DD PATH='<HFS-file-name>'

Note that your tickets will expire after a certain period of time based on your DCE system’s ticket expiration policy that is set up by your DCE administrator. If your client program inherits your login context, you should log in or reauthenticate yourself to DCE before the end of the Maximum Ticket Renewable Time for your system. See your DCE Administrator to find out this time. See z/OS DCE Administration Guide on how to reauthenticate yourself using the kinit command.

Also, suppose that you log into DCE as a principal and your DCE batch application inherits this principal’s context. If you log in again to renew or change your context to run subsequent jobs, the batch jobs currently running or in the queue will not know about this changed context. To eliminate any potential confusion over the context under which your application runs while in batch mode, your application should explicitly set up its own login context using the following series of calls:

- sec_login_setup_identity()
- sec_login_validate_identity()
- sec_login_set_context().

A programming example shows how this is done in “Greet with Name-Based Authorization” on page 117.

Notes:

1. Note that any applications that log into DCE using the above APIs or that performs a DCELOGIN in batch will overwrite your $HOME/krb5ccname file since a batch job in either the Shell or MVS environments shares the user’s home directory.

2. Ensure that any EUVSKRB5 data sets you create are RACF protected from other z/OS DCE users; otherwise, they can obtain the same permissions that you have by referencing your EUVSKRB5 data sets.
Inheriting Contexts for Multiple Programs

If you have a batch job that invokes multiple DCE programs, each program can inherit different login contexts by referencing different EUVSKRB5 data sets in your start up JCL, as shown in Figure 63:

```
//JOBNAME JOB (ACCOUNT),...your_job_parameters
/******************** **************************************************
//*                   JCL TO STARTUP MULTIPLE DCE JOBS                   *
//*                   ***********************************************
//JOB1 EXEC PGM=PROGRAM1,PARM='POSIX(ON)/ your_parameters'
//SYSOUT DD SYSOUT=g+O#SJ
//SYSPRINT DD SYSOUT=g+O#SJ
//SYSUDUMP DD SYSOUT=g+O#SJ
//STEPLIB DD DSN=USERPRFX.<APPLNM>.LOAD,DISP=SHR
//EUVSKRB5 DD DSN=dsn1,DISP=SHR
//******************** **************************************************
//JOB2 EXEC PGM=PROGRAM2,PARM='POSIX(ON)/ your_parameters'
//SYSOUT DD SYSOUT=g+O#SJ
//SYSPRINT DD SYSOUT=g+O#SJ
//CEEDUMP DD SYSOUT=g+O#SJ
//SYSDUMP DD DSN=USERPRFX.<APPLNM>.LOAD,DISP=SHR
//EUVSKRB5 DD DSN=dsn2,DISP=SHR
//******************** **************************************************
```

Figure 63. Starting Multiple DCE Applications in Batch

If you want to keep the login context the same for each of the programs, simply reference the same EUVSKRB5 data set in each job step.

Searching for Your Server

In "9. Starting the Greet Server" on page 72 the Greet server application prints its string binding after using the `rpc_binding_to_string_binding()` function. The components of the string binding are passed to the Greet client as input parameters during client start up. The Greet client uses the binding information to locate the Greet server from the RPC run time. In that example, you pass the complete string binding information, including:

- The supported RPC protocol
- The server’s host machine address
- The server’s endpoint.

Finding a server in this manner is cumbersome, especially for users running client applications, because they must find out the complete server address. Usually this information must be provided to users by an external method; perhaps they have to look in a file, or ask someone. The binding information may not always be available to users. Two methods to locate the server are introduced in this section to ease the search. These methods use the following RPC services:

1. DCE Host Daemon (DCE Host Daemon)
2. Name Service Interface (NSI).

For a detailed discussion on obtaining server binding information and managing bindings, refer to Chapter 6, "Binding" on page 203.
Using the DCE Host Daemon

Finding the server by using the DCE Host Daemon relieves the client application from the requirement of knowing the server’s endpoint. The client still requires binding information on the server’s protocol sequence and its host machine address. This information constitutes a partial binding. In Figure 64 on page 80, the DCE Host Daemon is used to locate the server endpoint and obtain a binding that is fully bound.

Modifying the Greet Server Code: You can find the source code for this version of the Greet example in the /usr/lpp/dce/examples/greet1 directory. The source files supplied with z/OS DCE for this example are as follows:

- `greet1.idl` (the IDL file)
- `greet1_client.c` (the client code)
- `greet1_server.c` (the server code)
- `greet1_manager.c` (the manager code)
- Makefile
- README.

If you cannot find the Greet1 example source files in the above directory, consult your systems programmer to find where they are located in your system.

The IDL file content is the same as before, except for the interface UUID. It is called `greet1.idl` for this example. To avoid any potential conflicts with other instances of the same Greet1 example and ensure that you are running a unique instance, you should generate your own interface UUID using the UUID Generator and replace the interface UUID in the IBM-supplied Greet1 IDL file with it.

In this version of the Greet program, make the following changes to the server code:

1. Replace the `rpc_server_use_all_protseq()` call in the previous Greet example with two calls: `rpc_network_is_protseq_valid()`, and `rpc_server_use_protseq()`. The `rpc_network_is_protseq_valid()` checks whether a specified transport protocol is supported by the RPC runtime library and the operating system. In this example, `ncadg_ip_udp` which represents connectionless or user datagram protocol/interface protocol (UDP/IP) is checked. The server then calls `rpc_server_use_protseq()` to obtain an endpoint on which to listen using the UDP/IP protocol. This protocol is hard-coded in this example.

2. To use the DCE Host Daemon to locate the Greet server endpoint, add the `rpc_ep_register()` call prior to the TRY, CATCH_ALL, ENDTRY code.

3. Because the code registers the server endpoint to the DCE Host Daemon, it should be unregistered when the server is shutdown in the CATCH_ALL macro:

The sample code in Figure 64 on page 80 shows the modified Greet1 server routine.
#pragma runopts(stack(12K,4K,ANY,KEEP))

#include <stdio.h>
#include <locale.h>
#include <dce/dce_error.h>
#include <dce/exc_handling.h>
#include "greet1.h"

#define MAX_CONCURRENT_CALLS 5

/* In the first part of the main function, the server calls 
the rpc_network_is_protseq_valid to check that its argument 
specifies a protocol sequence that is supported on its host 
both by the runtime library and the operating system. */

int main (int argc, char *argv[]) {

  rpc_binding_vector_p_t bvec;
  unsigned long st;
  int error_inq_st;
  dce_error_string_t error_text;
  idl_boolean validfamily;
  idl_char *string_binding;
  int i;

  setlocale(LC_ALL, "");

  if (argc != 1) {
    fprintf(stderr, "Usage: %s \n", argv[0]);
    fflush(stderr);
    exit(1);
  }

  validfamily = rpc_network_is_protseq_valid("ncadg_ip_udp", &st);
  if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot check protocol sequence - %s\n", error_text);
    fflush(stderr);
    exit(1);
  }

  if (!validfamily) {
    fprintf(stderr, "Protocol sequence is not valid\n");
    fflush(stderr);
    exit(1);
  }

  /\n  * Calling rpc_server_use_protseq to obtain an endpoint 
  on which to listen */

  rpc_server_use_protseq("ncadg_ip_udp", MAX_CONCURRENT_CALLS, &st);
  if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot use protocol sequence - %s\n", error_text);
    fflush(stderr);
    exit(1);
  }

  /\n  * Figure 64 (Part 1 of 3). Sample Code to Register the Greet1 Server Endpoint to the DCE Host Daemon */

  ...
Calling rpc_server_register_if to register its interface with the RPC runtime by suppying its interface specifier:

```
rpc_server_register_if(greet_v1_0_s_ifspec, NULL, NULL, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot register interface - %s\n", error_text);
    fflush(stderr);
    exit(1);
}
```

Calling rpc_server_inq_bindings to obtain a vector of binding handles that can be used to register the server's endpoint. The server then obtains, prints, and frees a string binding:

```
rpc_server_inq_bindings(&bvec, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot inquire bindings - %s\n", error_text);
    fflush(stderr);
    exit(1);
}
```

```
Bindings:
```

The server endpoint is registered in the local Endpoint Map:

```
rpc_ep_register(greet_v1_0_s_ifspec, bvec, (uuid_vector_p_t) NULL, (unsigned_char_p_t) "greet version 1.0 server", &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot register end point: %s\n", error_text);
    fflush(stderr);
    exit(1);
}
```

To begin listening for RPC requests, the server calls

```
rpc_server_listen. This call is placed within the TRY of a TRY, CATCH_ALL, ENDTRY sequence, so that if the server receives a signal while it is listening, it can unregister its interface and its endpoint before it exits.
```

---

Figure 64 (Part 2 of 3). Sample Code to Register the Greet1 Server Endpoint to the DCE Host Daemon
TRY {
    printf("Listening...
\n");
    fflush(stdout);
    rpc_server_listen(MAX_CONCURRENT_CALLS, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Error: %s\n", error_text);
        fflush(stderr);
        return (0);
    }
}

CATCH_ALL {  ──┐
    printf("Unregistering endpoint \n\n");  │
    fflush(stdout);  │
    rpc_ep_unregister(greet_v1_0_s_ifspec, bvec,
        (uuid_vector_p_t) NULL, &st);  │
    if (st != error_status_ok) {  │
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot unregister endpoint: %s\n",
            error_text);
        fflush(stderr);
        return (0);
    }
} ──┘

ENDTRY;

Figure 64 (Part 3 of 3). Sample Code to Register the Greet1 Server Endpoint to the DCE Host Daemon

The Greet1 client code, in Figure 65 on page 83 has been simplified to pass the string binding as one
start up parameter instead of three (the protocol sequence, host id, and endpoint) as in the first Greet
example. In this example, the rpc_string_binding_compose() function is not required and has been
removed.
Figure 65. Greet1 Client Code

z/OS DCE has implemented access control to the DCE Host Daemon that runs on z/OS. For this Greet server application and any DCE server application that accesses the DCE Host Daemon, the principal name (or the group to which it belongs) running the application requires the insert permission to register the server endpoint, and the delete permission to unregister the server endpoint. Server principals should use a special server permission that combines the insert and delete permission. There is also a Security group that is defined during the configuration of your machine which contains the necessary permissions to the DCE Host Daemon. Your server application inherits this group’s permissions to the DCE Host Daemon if it becomes a member of this group. See z/OS DCE Administration Guide for information on granting these permissions to DCE server principals.
The name service entry that points to the DCE Host Daemon running on a z/OS machine is
./hosts/<hostname>/config/epmap, where <hostname> is the DCE host name of interest. For example,
to list the ACL entries associated with the DCE Host Daemon that runs on a machine with host name
**bass** use the following command:
dccp -c acl show ./hosts/bass/config/epmap

Compile and link-edit the Greet1 server code as before, using the Makefile if you are building in the Shell,
or using JCL similar to that in "8. Building the Greet Client and Server Programs" on page 69 if you are
building in batch.

To start the Greet1 server in the Shell, in background mode, use the following command:
greet1_server &

To start up the Greet1 server in batch, use the example JCL shown in Figure 66.

```
// JOBNAME JOB (ACCOUNT),...your_job_parameters
//******************************************************************************
// JCL TO STARTUP THE GREET1 SERVER
//******************************************************************************
//GREET1SR EXEC PGM=GREET1SR,PARM='POSIX(ON)/'
/SYSOUT DD SYSOUT=*
/SYSPRINT DD SYSOUT=*
/SYSPRINT DD SYSOUT=*
/STEPLIB DD DSN=USERPRFX.GREET1.LOAD,DISP=SHR
```

*Figure 66. Starting the Greet1 Server in Batch Using the DCE Host Daemon*

When you check your output, you see the same output as before in Figure 55 on page 73, but with a
single different endpoint for the server binding (see Figure 67).

```
Bindings:
cnadg_ip_udp:9.21.6.97[1044]
Listening...
```

*Figure 67. Initial Greet1 Server Output*

**Note on Registering Servers:** You can register your server application’s endpoint information with
the DCE Host Daemon by using either the `rpc_ep_register()` or the `rpc_ep_register_no_replace()` API.
For servers running on connection-oriented RPC, the RPC runtime may open more than one socket,
resulting in more than one binding handle returned from the `rpc_server_inq_bindings()` call. The binding
handles returned differ only in their endpoint information.

The implication to your server applications running on connection-oriented RPC is that when you register
your endpoint to the DCE Host Daemon, you **must** use `rpc_ep_register_no_replace()` instead of
`rpc_ep_register()` in your code. If you use `rpc_ep_register()`, only the last binding handle information
received from the RPC runtime is registered. This situation does not occur for connectionless RPC, so
you can use `rpc_ep_register()` API.
Balancing Server Workload: For your application servers that register their dynamic endpoints with the DCE Host Daemon, the corresponding client applications rely on the RPC runtime and DCE Host Daemon to locate the compatible server.

For connection-oriented RPC, the RPC runtime internally calls `rpc_ep_resolve_binding()` to resolve a partially bound server binding handle to obtain a handle that is fully bound. If more than one compatible server instance is registered in the local endpoint map, this routine randomly selects one server to achieve the load balance.

For connectionless RPC, the DCE Host Daemon always selects the first compatible server instance from the endpoint map database. This may result in this server instance becoming overworked while other compatible server instances are idle. To minimize this workload imbalance, use object UUIDs to specify specific server instances. For more information on using object UUIDs, refer to "Using Object UUIDs to Avoid Binding Ambiguity" on page 225.

Modifying the Greet Client Start up: To start the Greet1 client, pass the transport protocol and the server’s host identifier as one argument to the client program. This constitutes a partial binding as there is no endpoint component. You do not pass the endpoint argument to the Greet client at start up as in the previous example. Because the Greet server registers its endpoint to the DCE Host Daemon, the DCE Host Daemon provides the endpoint information enabling the Greet client to find its compatible server. No special permissions are required to read from the DCE Host Daemon end point map, therefore the Greet client can run unauthenticated.

To start the Greet1 client in the Shell, use the following command:

```
greet1_client ncadg_ip_udp:9.21.6.97[1gz'ro@ot44] 1
```

To start the Greet1 client in batch, use the sample JCL as shown in Figure 68.

```
//JJobname JOB (ACCOUNT)
//**********************************************************************************************
//*
//* JCL TO STARTUP THE GREET1 CLIENT
//*
//**********************************************************************************************
//GREET1 EXEC PGM=GREET1CL,PARM='POSIX(ON)/ ncadg_ip_udp:9.21.6.97 1044 1'
//SYSOUT DD SYsOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//STEPLIB DD DSN=USERPRFX.GREET1.LOAD,DISP=SHR
```

Figure 68. Greet1 Client Start up JCL

The Greet1 client output appears as in Figure 69.

```
Bound to ncadg_ip_udp:9.21.6.97[1044]  
The Greet1 Server said: Hi client!
```

Figure 69. Greet1 Client Output

The Greet1 server output appears as in Figure 70 on page 86.
Using the Name Service Interface

The easiest method of locating a server binding is to use the RPC name service interface. This is the interface to the Cell Directory Service (CDS), the database used to store server names in a cell. Using CDS in a client application removes the requirement of knowing the server’s host address and protocol sequence. And, if the DCE Host Daemon is used along with CDS, its end point is also not required. The server is accessed with an easy to remember entry name, as opposed to a cryptic string binding. This access requires additional DCE function calls in both your client and server application. For more information on using the NSI, refer to Chapter 7, “Using the DCE Name Service” on page 215.

Usually, a server exports binding information for one or more of its interface identifiers and its object UUIDs, if any, to an entry in the CDS database. The name service entry to which a server exports binding information is known as a server entry. Clients search for exported binding information associated with an interface and object. Distributed applications use the name service to place, inquire and manage information about server entries.

In this example, the Greet server’s binding information is stored in the CDS database.

You can find the source code for this version of the Greet example in the /usr/lpp/dce/examples/greet2 directory. The source files supplied with z/OS DCE for this example are as follows:

- greet2.idl (the IDL file)
- greet2_client.c (the client code)
- greet2_server.c (the server code)
- greet2_manager.c (the manager code)
- Makefile
- README.

If you cannot find the Greet2 example source files in the above directory, consult your systems programmer to find where they are located in your system.

The IDL file content is the same as before, except for the interface UUID. It is called greet2.idl for this example. To avoid any potential conflicts with other instances of the same Greet2 example and ensure that you are running a unique instance, you should generate your own interface UUID using the UUID Generator and replace the interface UUID in the IBM-supplied Greet2 IDL file with it.

The following section shows you how to modify the Greet application to export and import the Greet server binding to and from the CDS database.
Exporting the Greet Server Binding to the Namespace: The only major changes you make to the Greet server are shown by the following reference keys in Figure 71 on page 88:

1. Declare a new variable entry_name, which stores the name of the Greet server entry. It is passed as parameter to the Greet server during start up.

2. Export the binding received from the RPC runtime to the CDS namespace using the rpc_ns_binding_export() function. The export to the namespace must occur after the server endpoint is registered in the local end point map.

   Note: The APIs for exporting bindings to the namespace require a login context for your server. Normally the server inherits this context when you log into DCE. Consult z/OS DCE Administration Guide for more information on logging into DCE.

3. Unexport the binding from the namespace from within the TRY, CATCH ALL, and ENDTry sequence to ensure that the namespace entry is deleted prior to termination of the Greet server application in the event of an error.

There are no changes required to the Greet manager code. Figure 71 on page 88 shows the Greet server source code modified to use the RPC name service interface to CDS.
#pragma runopts(stack(12K,4K,ANY,KEEP))

#include <stdio.h>
#include <locale.h>
#include <dce/dce_error.h>
#include <dce/exc_handling.h>
#include "greet2.h"

#define MAX_CONCURRENT_CALLS 5

/* In the first part of the main function, the server calls
the rpc_network_is_protseq_valid to check that its argument
specifies a protocol sequence that is supported on its host
both by the runtime library and the operating system. */

int main (int argc, char *argv[])
{
    rpc_binding_vector_p_t bvec;
    unsigned long st;
    int error_inq_st;
    dce_error_string_t error_text;
    idl_boolean validfamily;
    idl_char *string_binding;
    char *entry_name;
    int i;

    setlocale(LC_ALL, "");

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <server_entry_name>\n", argv[0]);
        fflush(stderr);
        exit(1);
    }

    entry_name = argv[1];

    validfamily = rpc_network_is_protseq_valid("ncadg_ip udp", &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st,error_text,&error_inq_st);
        fprintf(stderr, "Cannot check protocol sequence - %s\n", error_text);
        fflush(stderr);
        exit(1);
    }

    if (!validfamily) {
        fprintf(stderr, "Protocol sequence is not valid\n";
        fflush(stderr);
        exit (1);
    }

    /* Calling rpc_server_use_protseq to obtain an endpoint
on which to listen */

Figure 71 (Part 1 of 4). Greet2 Server Source Code — Modified for CDS.
```c
rpc_server_use_protseq("ncadg_ip_udp", MAX_CONCURRENT_CALLS, &st)
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    printf("Cannot use protocol sequence - %s\n", error_text);
    fflush(stdout);
    exit(1);
}

/* Calling rpc_server_register_if to register its interface with */
the RPC runtime by supplying its interface specifier */
rpc_server_register_if(greet_v1_0_s_ifspec, NULL, NULL, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf("Cannot register interface - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* Calling rpc_server_inq_bindings to obtain a vector of */
/* binding handles that can be used to register the server's */
/* string binding */
rpc_server_inq_bindings(&bvec, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot inquire bindings - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

printf("Server %s bindings:\n", entry_name);
fflush(stdout);
for (i = gz'ro@ot; i < bvec->count; i++) {
    rpc Binding_to_string_binding(bvec->binding_h[i],
        &string_binding, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot convert to string binding: %s\n", error_text);
        fflush(stderr);
        exit(1);
    }
    printf("%s\n", (char *)string_binding);
    fflush(stdout);
    rpc_string_free(&string_binding, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot free string memory: %s\n", error_text);
        fflush(stderr);
        exit(1);
    }
}
```

Figure 71 (Part 2 of 4). Greet2 Server Source Code — Modified for CDS.
/ The server endpoint is registered in the local Endpoint Map */

rpc_ep_register(greet_v1_0_s_ifspec, bvec,
    (uuid_vector_p_t) NULL,
    (unsigned_char_p_t) "greet version 1.0 server",
    &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot register end point: %s\n", error_text);
        fflush(stderr);
        exit(1);
    }

/* export the binding vector the runtime gave us to the namespace */

rpc_ns_binding_export(rpc_c_ns_syntax_dce, entry_name,
    greet_v1_0_s_ifspec, bvec, (uuid_vector_t) NULL, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot export binding vector: %s\n", error_text);
        fflush(stderr);
        exit(1);
    }

/* To begin listening for RPC requests, the server calls */
rpc_server_listen. This call is placed within the TRY of a
TRY, CATCH_ALL, ENDTRY sequence, so that if the server receives
a signal while it is listening, it can unexport its entry from
the namespace and unregister its interface before it exits. */

TRY {
    printf("Server %s is listening...\n", entry_name);
    fflush(stdout);
    rpc_server_listen(MAX_CONCURRENT_CALLS, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Error: %s\n", error_text);
        fflush(stderr);
    }
}

CATCH_ALL {
    CLEANUP:
    /* unexport binding vector from namespace --
        not usually done for a persistent server */
    fprintf(stdout, "Server %s unexporting\n", entry_name);
    fflush(stdout);
    rpc_ns_binding_unexport(rpc_c_ns_syntax_dce,
        entry_name, greet_v1_0_s_ifspec,
        (uuid_vector_t *) NULL, &st);
}

Figure 71 (Part 3 of 4). Greet2 Server Source Code — Modified for CDS.
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text,&error_inq_st);
    fprintf(stderr, "Cannot unexport binding vector: %s
", error_text);
    fflush(stderr);
}

printf("Unregistering endpoint \n");
fflush(stdout);

rpc_ep_unregister(greet_v1_0_s_ifspec, bvec,
    (uuid_vector_p_t) NULL, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text,&error_inq_st);
    fprintf(stderr, "Cannot unregister endpoint: %s
", error_text);
    fflush(stderr);
}

ENDTRY;
return(0);

---

Figure 71 (Part 4 of 4). Greet2 Server Source Code — Modified for CDS.

Importing the Greet2 Server Binding from the Namespace

The only major changes you make to the Greet2 client routine to import the Greet2 server binding is shown in Figure 72 on page 92 by the following reference keys:

1. Declare a new variable `server_name`, which is used to store the name of the Greet2 server entry. The client uses this name to locate the Greet server in the CDS namespace.
2. Declare a new variable `import_context` required for RPC to track the import context internally.
3. Call the import routines to obtain a binding handle for the Greet2 server as following:

   ```c
   rpc_ns_binding_import_begin()
   rpc_ns_binding_import_next()
   rpc_ns_binding_import_done()
   ```

   Figure 72 on page 92 shows the Greet client source code modified to use the RPC name service interface to CDS.
```c
#include <stdio.h>
#include <locale.h>
#include <dce/dce_error.h>
#include "greet2.h"

int main(int argc, char* argv[]) {
    handle_t h;
    unsigned long st;
    int error_inq_st;
    dce_error_string_t error_text;
    idl_char string_binding;
    int i, MAX_PASS;
    char reply[STR_SZ],
    *server_name;
    rpc_ns_import_handle_t import_context;

    setlocale(LC_ALL, "");

    if (argc != 3) {
        fprintf(stderr, "Usage: %s <server_name> <passes>\n", argv[0]);
        fflush(stderr);
        exit (1);
    }

    server_name = argv[1];

    /* import compatible server bindings from the namespace */
    rpc_ns_binding_import_begin(rpc_c_ns_syntax_dce,
        server_name, greet_v1_0_c_ifspec,
        (uuid_t *)NULL, &import_context, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st,error_text,&error_inq_st);
        fprintf(stderr, "Cannot begin importing binding: %s\n",error_text);
        fflush(stderr);
        exit(1);
    }

    /* sift through bindings and choose the first one over udp */

    while (1) {
        rpc_ns_binding_import_next(import_context, &h, &st);
        if (st == rpc_s_no_more_bindings) {
            dce_error_inq_text(st,error_text,&error_inq_st);
            fprintf(stderr, "Cannot find binding over udp: %s\n",error_text);
            fflush(stderr);
            exit(1);
        }
    }
}
```

*Figure 72 (Part 1 of 2). Greet2 Client Source Code - Modified for CDS*
rpc_binding_to_string_binding(h, &string_binding, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot convert binding to string binding: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* out of curiosity, print the binding */

if (strstr(string_binding, "ncadg_ip_udp") != 0) {
    fprintf(stdout, "Client bound to server %s at %s\n", server_name, string_binding);
    fflush(stdout);  
    rpc_string_free(&string_binding, &st);
    break;
}

rpc_string_free(&string_binding, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot free string memory: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* end the binding import lookup loop */

rpc_ns_binding_import_done(&import_context, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot end binding import: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

fprintf(stdout, "\n");
fflush(stdout);
MAX_PASS=atoi(argv[2]);
fflush(stdout);

for (i=1; i <= MAX_PASS; i++) {
    greet_rpc(h, "Hello Server!", reply);
    printf("The Greet Server said: %s\n", reply);
    fflush(stdout);
}
return(0);

Figure 72 (Part 2 of 2). Greet2 Client Source Code - Modified for CDS

Compile and link-edit the modified Greet client and server as in "8. Building the Greet Client and Server Programs" on page 69
Starting the Greet2 Server and Client

Start the Greet client and server as you did in "9. Starting the Greet Server" on page 72 and "10. Starting the Greet Client" on page 73. In this instance, you modify the parameters passed to both the client and server applications during their start up.

**Note:** Before you start up the Greet server, ensure that the following processes are running in your DCE cell:

- DCE Host daemon (dced)
- Security daemon (SECD)
- CDS advertiser (CDSADV)
- CDS clerk (CDSCLRK)
- CDS daemon (CDSD).

You can display the active status of any DCE processes running on your machine using the SDSF 'Display active users of the system' panel. For information on processes running on other machines in your cell, consult your DCE administrator.

**Starting the Greet2 Server for CDS:** As before in "Using the DCE Host Daemon" on page 79, you must log into DCE but in this example, the principal under which your server application runs requires *insert* access to the parent directory in the CDS namespace to which it exports its entry object. Because the principal that runs your server application creates the object `greet2` in the namespace, it possesses *read* and *write* permission to that object by default, enabling it to manipulate the object.

For the server, pass its entry name, `/./<your_dir_name>/greet2`, as an argument to the Greet server program using the GPARM= statement. The `/./` indicates that `your_dir_name/greet2` is the cell-relative part of the Greet server's name service entry name. The parent directory, represented by `/<your_dir_name>`, must exist in the CDS namespace so you can export the leaf entry, that is, the object `greet2`, into it. Choose a directory name that is meaningful to you and have your DCE administrator create this directory in the CDS namespace, as well as granting your principal the required *insert* permission to this directory. This name, `greet2`, is exported into the CDS namespace by the Greet server.

To start the Greet2 server in the Shell, use the following command:

```
greet2_server /./<your_dir_name>/greet2
```

To start the Greet2 server in batch, use sample JCL for starting up the server in Figure 73.

```bash
//JOBNAME JOB (ACCOUNT),...your_job_parameters
//*******************************************************************************
//
// JCL TO STARTUP THE GREET2 SERVER
//
//*******************************************************************************
//GREET2SR EXEC PGM=GREET2SR,PARM='POSIX(ON)/ ./.<your_dir_name>/greet2'
//SYSOUT DD SYSOUT=g+O#SJ
//SYSPRINT DD SYSOUT=g+O#SJ
//SYSUDUMP DD SYSOUT=g+O#SJ
//STEPLIB DD DSN=USERPRFX.GREET2.LOAD,DISP=SHR
```

*Figure 73. Starting the Greet2 Server in Batch Using CDS*

After you have started the server, check the server output. It should be similar to that in *Figure 74 on page 95*. 

---

94  DCE Application Development Guide: Introduction and Style
Prior to starting up your client, log into DCE once more, but as a different principal than for the server application. The principal under which your client runs requires read permission to the CDS namespace. Consult your DCE Administrator to ensure that this principal has read permission to the greet2 object in the namespace. In this example the input parameter to the client routine has changed to the server entry name — /./:<your_dir_name>/greet2, instead of the string binding as shown in Figure 58 on page 74.

To start the Greet2 client in the Shell, use the following command:

```
greet2_client /./:<your_dir_name>/greet2 1
```

To start the Greet2 client in batch, use the sample JCL shown in Figure 75.

```
//JOBNAME JOB (ACCOUNT),...your_job_parameters

//******************************************************************************
//*
//* JCL TO STARTUP THE GREET2 CLIENT
//*
//*
//GREET2CL EXEC PGM=GREET2CL,PARM='/./:<your_dir_name>/greet 2'
//SYSOUT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//STEPLIB DD DSN=USERPRFX.GREET2.LOAD,DISP=SHR
```

After the Greet2 client application terminates, check its output. You should see output similar to that in Figure 76.

```
Client bound to server /./:<your_dir_name>/greet2 at ncadg_ip_udp:9.21.6.97[1131]
The Greet Server said: Hi client !
```

Checking the Greet2 server output once more, you should see output similar to that in Figure 77.

```
Server /./:<your_dir_name>/greet2 bindings:
ncadg_ip_udp:9.21.6.97[1131]
Server /./:<your_dir_name>/greet2 is listening...
The client says: Hello Server !
```
Monitoring Your Distributed Application

To monitor your distributed application and associated DCE processes in real time, use the z/OS TCP/IP Real Time Network Monitor. This monitor displays all the TCP/IP network connections of the various applications running on your z/OS host. You can view the connections as they are assigned to your application, and watch communications as they occur across the network between various DCE entities such as the DCE Host Daemon, CDS Advertiser and Clerk, and your distributed application. This step is especially useful when you are monitoring the start up of your DCE application.

To invoke a dynamic version of the network monitor, enter the following command from any ISPF command line:

```
tso netstat int 2
```

The `int 2` is the *interval* parameter that specifies an update of the monitor output. In the above example, the monitor output is refreshed every 2 seconds. For complete information about using the NETSTAT command, refer to [z/OS Communications Server: IP User's Guide](https://www.ibm.com/support/docview.wss?uid=swg21428780). You see a screen similar to that contained in Figure 78.

![Figure 78. Network Monitor Output - Dynamic Update](image)

<table>
<thead>
<tr>
<th>User Id</th>
<th>B Out</th>
<th>B In</th>
<th>L Port</th>
<th>Foreign Socket</th>
<th>State</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>FTPSERVE</td>
<td>0</td>
<td>0</td>
<td>FTP-C</td>
<td><em>.</em></td>
<td>Listen</td>
<td>122:21:06</td>
</tr>
<tr>
<td>INTCLIEN</td>
<td>0</td>
<td>0</td>
<td>TELNET</td>
<td><em>.</em></td>
<td>Listen</td>
<td>122:21:14</td>
</tr>
<tr>
<td>OMVS</td>
<td>21595</td>
<td>22729</td>
<td>135</td>
<td><em>.</em></td>
<td>UDP</td>
<td>0:00:01</td>
</tr>
<tr>
<td>OMVS</td>
<td>864</td>
<td>704</td>
<td>1116</td>
<td><em>.</em></td>
<td>UDP</td>
<td>0:00:01</td>
</tr>
<tr>
<td>OMVS</td>
<td>244912</td>
<td>146390</td>
<td>1272</td>
<td><em>.</em></td>
<td>UDP</td>
<td>0:28:50</td>
</tr>
<tr>
<td>OMVS</td>
<td>32628</td>
<td>101848</td>
<td>1273</td>
<td><em>.</em></td>
<td>UDP</td>
<td>0:28:50</td>
</tr>
<tr>
<td>OMVS</td>
<td>0</td>
<td>88</td>
<td><em>.</em></td>
<td>UDP</td>
<td>0:28:50</td>
<td></td>
</tr>
<tr>
<td>OMVS</td>
<td>0</td>
<td>1940</td>
<td><em>.</em></td>
<td>UDP</td>
<td>0:26:18</td>
<td></td>
</tr>
<tr>
<td>OMVS</td>
<td>340</td>
<td>905</td>
<td>1940</td>
<td>TLBDSBME..1133</td>
<td>Established</td>
<td>0:19:39</td>
</tr>
<tr>
<td>OMVS</td>
<td>5694</td>
<td>5392</td>
<td>1130</td>
<td><em>.</em></td>
<td>UDP</td>
<td>0:00:00</td>
</tr>
</tbody>
</table>

Refresh interval: 2 Seconds. TCB's In Use: 3

In the above example, the DCE daemons running on your machine, such as `dced`, CDS Clerk, and CDS Advertiser are represented by user ID OMVS.

When you start the Greet2 server and client applications, and invoke the NETSTAT monitor, you see a number of ports established for your application, similar to Figure 79 on page 97. In this example, the Greet2 server is denoted by User ID `GREET2SR`; the Greet2 client is denoted by User ID `GREET2CL`.

---

96  DCE Application Development Guide: Introduction and Style
Using Automatic Binding Handles

There are several methods for a client to manage binding information for its remote procedure calls. The most common methods include: explicit, implicit, and automatic. All the examples presented so far use explicit binding.

The different binding management methods are discussed in detail in "Binding Methods" on page 209.

The automatic method is the simplest way for a client to manage binding information. With automatic binding, the server exports its binding information to a name service database, and the client stub automatically manages a binding for the application code. Binding import operations are conducted by the client stub code.

Note that the automatic binding method only works in an unauthenticated mode.

You can find the source code for this version of the Greet example in the /usr/lpp/dce/examples/greet3 directory. The source files supplied with z/OS DCE for this example are as follows:

- greet3.idl (the IDL file)
- greet3.acf (the IDL file)
- greet3_client.c (the client code)
- greet3_server.c (the server code)
- greet3_manager.c (the manager code)
- Makefile
- README.

If you cannot find the Greet3 example source files in the above directory, consult your systems programmer to find where they are located in your system.
The IDL file is called `greet3.idl` for this example. To avoid any potential conflicts with other instances of the same Greet3 example and ensure that you are running a unique instance, you should generate your own interface UUID using the UUID Generator and replace the interface UUID in the IBM-supplied Greet3 IDL file with it.

This example includes an ACF file which specifies that the automatic binding method is to be used.

**Modifying the Greet Code**

To use automatic binding handles for the Greet client, make the following changes to the Greet client code:

1. Specify the `auto_handle` attribute in `greet3.acf` (or `USERPRFX.ACF(GREET3)` if you are using a PDS). Note that the explicit binding handle must be removed from the IDL file for this attribute to take effect. See the sample code in Figure 80.

```
[auto_handle] interface greet
{
}
```

*Figure 80. ACF File for the Greet3 Application — auto_handle*

2. Remove the primitive binding handle `h` from the IDL file, `greet3.idl` (or `USERPRFX.IDL(GREET3)` if you are using a PDS). Your IDL file should look similar to Figure 81.

```
[uuid(2924413-4143-19ea-a6e0-000dce1d345),
version(1.0)]

/* This operation has no binding handle parameter,
   therefore it uses automatic binding */

interface greet
{
    const short int STR_SZ=128;
    void greet_rpc ( 
        [in] char client_greeting[STR_SZ],
        [out] char server_reply[STR_SZ]
    );
}
```

*Figure 81. IDL File for the Greet3 Application — Using Automatic Binding*

3. Compile the IDL and ACF files.

4. Remove the primitive binding handle `h` from `greet3_manager.c` (`USERPRFX.GREET3.C(MANAGER)` if you use a PDS). Your manager should look similar to Figure 82 on page 99.
#include <stdio.h>
#include "greet3.h"

void greet_rpc(char *client_greeting,
               char *server_reply)
{
    printf("The client says: %s
", client_greeting);
    fflush(stdout);
    strncpy(server_reply, "Hi client!", STR_SZ);
}

Figure 82. Manager Code for the Greet3 Application — Using Automatic Binding

5. Remove or comment out all code involving binding handle h, the server entry name, and the import operations loop in your client routine. Your client routine should be similar to Figure 83.

#include <stdio.h>
#include <locale.h>
#include <dce/dce_error.h>
#include "greet3.h"

int main(int argc, char* argv[])
{
    error_status_t st, error_inq_st;
    int i, MAX_PASS;
    char reply[STR_SZ];

    setlocale(LC_ALL, "");

    if (argc != 2) {
        fprintf(stderr, "Usage: %s passes
", argv[0]);
        fflush(stderr);
        exit (1);
    }

    snprintf(argv[1], STR_SZ, "%s", argv[1]);
    MAX_PASS= atoi(argv[1]);

    for (i=1; i <= MAX_PASS; i++) {
        greet_rpc("Hello Server!", reply);
        printf("The Greet Server said: %s
", reply);
        fflush(stdout);
    }

    return(0);
}

Figure 83. Client Code for Greet3 Application — Using Automatic Binding

Note the relative simplicity and compactness of the client routine in Figure 83 when compared to the example in Figure 72 on page 92. Check the Greet3 client stub code. Note that the task of finding the server has been transferred from the client routine to the client stub. Compile the Greet3 client stub and source code once again.

The Greet3 server code is the same as the Greet2 server code, except that the greet2.h header file is included instead of greet2.h. Start the Greet3 server the same way as the Greet2 server.
In this example, when you run the Greet3 client, you must supply your server name using the **RPC_DEFAULT_ENTRY** environment variable, so your client knows what server to search for. To set this variable, you should modify your home `envar` file to contain:

```
RPC_DEFAULT_ENTRY=/.:/<your_dir_name>/greet3
```

For other methods to set this environment variable, see Step 5. Setting Environment Variables on page 53.

To start the Greet3 client in the Shell, use the following command:

```
greet3_client 1
```

To start the Greet3 client in batch, use the sample JCL shown in Figure 84.

```
//JOBNAME JOB (ACCOUNT)
//**********************************************************************************************
// *
// * JCL TO STARTUP THE GREET3 CLIENT
// *
//**********************************************************************************************
//GREET3CL EXEC PGM=GREET3CL,PARM='POSIX(ON)/ 1'
//SYSOUT DD SYSOUT=
//SYSPRINT DD SYSOUT=
//SYSUDUMP DD SYSOUT=
//STEPLIB DD DSN=USERPRFX.GREET3.LOAD,DISP=SHR
```

Figure 84. Greet3 Client Start up JCL

**Adding Multiple Object Types**

Until now, you have used a single operation in the Greet manager routine. You can define more than one operation for your interface and have it run based on object types. For example, suppose there are ten printers at your location, numbered PRINTER1 through PRINTER10, that correspond to objects. Say half are line printers, and half are laser printers. You could write a print interface with two operations or implementations: one for the laser printers and one for the line printers. The two operations would correspond to the object types. DCE enables you to invoke one of the two implementations of the print interface depending on the object type that is registered with the RPC runtime. See Assigning Types to Objects section in the z/OS DCE Application Development Guide: Core Components for more information on objects and object types.

In this next example, you apply some of the object-oriented features of DCE to the Greet application. You redefine the Greet interface to include three object types that correspond to three languages used in the Greet communication exchange. That is, the greetings that are exchanged will be in one of the three languages (or object types) depending on the nationality (or object). In this example, the languages are:

- English
- French
- Spanish.

There are also seven nationalities, each of which corresponds to an object. The seven nationalities (or objects) are:

- Canadian
- British
- American
- French
• Belgian
• Mexican
• Spanish.

In the example, the user specifies nationality or object using a nationality code. Based on the nationality
code, one of the three Greet interface implementations is invoked. This is set up using a series of
`rpc_object_set_type()` calls to map the object to its object type. At run time, the Greet implementation
corresponding to the type is invoked. Following is the mapping of objects to the object type in this
example.

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>Canadian</td>
</tr>
<tr>
<td>English</td>
<td>American</td>
</tr>
<tr>
<td>English</td>
<td>British</td>
</tr>
<tr>
<td>French</td>
<td>French</td>
</tr>
<tr>
<td>French</td>
<td>Belgian</td>
</tr>
<tr>
<td>Spanish</td>
<td>Spanish</td>
</tr>
<tr>
<td>Spanish</td>
<td>Mexican</td>
</tr>
</tbody>
</table>

You can find the source code for this version of the Greet example in the `/usr/lpp/dce/examples/greet4` directory. The source files supplied with z/OS DCE for this example are as follows:

- `greet4.idl` (the IDL file)
- `greet4_client.c` (the client code)
- `greet4_server.c` (the server code)
- `greet4_manager.c` (the manager code)
- Makefile
- README.

If you cannot find the Greet4 example source files in the above directory, consult your systems
programmer to find where they are located in your system.

To avoid any potential conflicts with other instances of the same Greet4 example and ensure that you are
running a unique instance, you should generate your own interface UUID using the UUID Generator and
replace the interface UUID in the IBM-supplied Greet4 IDL file with it.

In this example, explicit binding handling is used. This example demonstrates the flexibility you have
when using explicit binding handles. Because this version of Greet uses CDS and DCE Host Daemon,
you need to make the following changes to the version of the Greet application you created in "Using the
Name Service Interface" on page 86.

1. IDL file: Define the seven object UUID string constants.
2. Manager: Code the three implementations of the Greet Interface and the corresponding Entry Point
   Vectors (EPV).
3. Server: Declare and generate the necessary object and type UUIDs. Map the objects to the correct
   object type, registering the interfaces to the RPC runtime and the DCE Host Daemon.
4. Client: Declare and generate the required object UUIDs and make the RPC call.
Modifying the Greet IDL File

Figure 85 shows the IDL file for the Greet application. Here you add seven constant character strings corresponding to the object UUIDs for the seven nationalities. See reference key 1. You derive the seven character strings by running the UUIDGEN utility with the number_of_uuid_strings argument set to 7. Enter the following command in either the Shell or TSO/E environment:

```
uuidgen -n 7
```

Copy the seven UUID strings that are output to the screen into the IDL file for your application. These UUIDs are unique throughout DCE because they are generated using UUIDGEN. After modifying your IDL file, compile it using the IDL compiler as in "5. Compiling the Greet Interface with the IDL Compiler" on page 61, but include the no_mepv IDL compiler option.

With this option, the IDL compiler does not generate a manager entry point vector (EPV) in the server stub. In this Greet example, you specify three manager EPVs, {greet_english}, {greet_french}, and {greet_spanish} in the Greet manager code. Each EPV corresponds to exchanging the greeting in a different language. In the Greet server, you register all three manager EPVs to the RPC runtime along with the Greet interface specifier, greet_v1_0_s_ifspec. All of this, plus a series of rpc_object_set_type() calls in the server code enables the server to determine which operation it runs when the Greet client makes the greet_rpc() remote call. Note that in the Greet4 client code, the remote procedure call name does not need to change for a different operation to run. The server determines which of the three operations it runs based on the object UUID specified in the binding handle from the client's RPC, by mapping greet_rpc() to one of the three EPVs.

In previous Greet examples, you entered null for the manager EPV field when you registered the Greet interface. In those cases, the RPC runtime searched and found the manager EPV generated in the server stub.

The seven string UUID constants are defined in the client and the server routines by including the header file that is generated.

```c
[uuid(20975713-4143-19ea-a6e0-0000dce12345),
 version(1.0)
]

interface greet
{
    const short int STR_SZ=128;
    const char *CANADIAN_OBJECT = "d58ab108-b3c6-11ca-891c-c9c2d4ff3b52";
    const char *BELGIAN_OBJECT = "d58ab208-b3c6-11ca-891c-c9c2d4ff3b52";
    const char *FRENCH_OBJECT = "d58ab308-b3c6-11ca-891c-c9c2d4ff3b52";
    const char *AMERICAN_OBJECT = "d58ab408-b3c6-11ca-891c-c9c2d4ff3b52";
    const char *BRITISH_OBJECT = "d58ab508-b3c6-11ca-891c-c9c2d4ff3b52";
    const char *MEXICAN_OBJECT = "d58ab608-b3c6-11ca-891c-c9c2d4ff3b52";
    const char *SPANISH_OBJECT = "d58ab708-b3c6-11ca-891c-c9c2d4ff3b52";

    void greet_rpc (
        [in]  handle_t h,
        [in]  char cT|ient_greeting[STR_SZ],
        [out] char server_reply[STR_SZ]
    );
}
```

Figure 85. Greet4 IDL File — for Multiple Object Types
Modifying the Greet4 Manager

Figure 86 shows the three implementations of the Greet interface in English, French, and Spanish. The operation is the same as before, only now it takes place in the three languages instead of one. It is necessary to define an EPV corresponding to each implementation.

The EPVs in the Greet example are identified by the following reference keys:

1. `greet_v1_0_english`
2. `greet_v1_0_french`
3. `greet_v1_0_spanish`

Modify the Greet4 manager routine as shown in Figure 86.

```c
#include <stdio.h>
#include "greet4.h"

/* English */
void greet_english(handle_t h,
    char *client_greeting,
    char *server_reply)
{
    printf("The client says: %s\n", client_greeting);
    fflush(stdout);
    strncpy(server_reply, "Hi client!", STR_SZ);
}

globaldef greet_v1_0_epv_t greet_v1_0_english = {greet_english};

/* French */
void greet_french(handle_t h,
    char *client_greeting,
    char *server_reply)
{
    printf("Le client parle: %s\n", client_greeting);
    fflush(stdout);
    strncpy(server_reply, "Bonjour client!", STR_SZ);
}

globaldef greet_v1_0_epv_t greet_v1_0_french = {greet_french};

/* Spanish */
void greet_spanish(handle_t h,
    char *client_greeting,
    char *server_reply)
{
    printf("El cliente dice: %s\n", client_greeting);
    fflush(stdout);
    strncpy(server_reply, "Buenas Dias client!", STR_SZ);
}

globaldef greet_v1_0_epv_t greet_v1_0_spanish = {greet_spanish};
```

Figure 86. Greet4 Manager — for Multiple Object Types
Modifying the Greet4 Server

The modifications you make to the Greet4 server routine in Figure 87 on page 105 are explained with the following reference keys:

1. Declare the three EPVs, the three type UUIDs, seven object UUIDs, and a vector of object UUIDs required for this application.
2. Generate three type UUIDs corresponding to the three different languages using the `uuid_create()` function.
3. Generate seven object UUIDs from the object strings you defined in the IDL file. Each object UUID corresponds to the seven nationalities.
4. Register the Greet4 server interface three times, once for each object type UUID and the corresponding EPV. With only one object type in the past examples, you registered nulls for both the object type UUID and EPV.
5. Call the `rpc_object_set_type()` routine seven times to map each of the seven objects (or nationalities) to one of the three object types (or languages).
6. Create a vector of seven object UUIDs, which is registered in the local EPM and exported to CDS at server start up.
7. Modify the `rpc_ep_register()` call, which registers the end point to the DCE Host Daemon to include the vector of object UUIDs.
8. Modify the `rpc_ns_binding_export()` call, which exports the binding vector to the namespace to also include the vector of object UUIDs.
9. Similarly, modify the `rpc_ns_binding_unexport()` and `rpc_ep_unregister()` routines to include the vector of object UUIDs.

In this scenario, the object UUIDs generated in the client and server (from the character string constants declared in the IDL file) are the link to the Greet4 implementation that is invoked. Linking is done by the `rpc_object_set_type()`, which maps the object UUID to the object type UUID. The type UUID is linked to the correct EPV corresponding to the implementation when the interface is registered. Note that the interface is registered three times to register each type UUID and its corresponding EPV.

In the previous versions of Greet, there was no need to assign object types because there was only one manager implementation. In those instances, the null object and type UUIDs were used when registering the Greet server interface and the endpoint. Also, the null binding vector was exported to the CDS namespace.
/* The server declares the manager EPVs defined in the
   manager.c module. There are three EPVs in this example, one
   for each operation TYPE coded in the manager routine. */

extern greet_v1_0_epv_t greet_v1_0_english;
extern greet_v1_0_epv_t greet_v1_0_french;
extern greet_v1_0_epv_t greet_v1_0_spanish;

/* In the first part of the main function, the server assigns
   its name (read in as an input parameter) to variable entry_name.
   The rpc_network_is_protseq_valid function checks that its argument
   specifies a protocol sequence that is supported on its host
   both by the runtime library and the operating system. */

int main (int argc, char *argv[]) {
  rpc_binding_vector_p_t bvec;
  dce_error_string_t error_text;
  unsigned long st;
  int error_inq_st;
  idl_boolean validfamily;
  idl_char *string_binding;
  char *entry_name;
  int i;
  uuid_t uuid_english, uuid_french, uuid_spanish;
  uuid_t canadian, british, american, french,
        belgian, mexican, spanish;
  typedef struct {
    unsigned32 count;
    uuid_t uuid[7];
  } UUID_VECTOR;
  UUID_VECTOR obj_uuids;
  setlocale(LC_ALL, "");

  if (argc != 2) {
    fprintf(stderr, "Usage: %s <entry_name>\n", argv[0]);
    fflush(stderr);
    exit(1);
  }

  entry_name = argv[1];

  validfamily = rpc_network_is_protseq_valid("ncadg_ip_udp", &st);
  if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot check protocol sequence - %s\n", error_text);
    fflush(stderr);
    exit(1);
  }

Figure 87 (Part 1 of 6). Greet4 Server Code — Modified for Multiple Object Types
if (!validfamily) {
    fprintf(stderr, "Protocol sequence is not valid\n");
    fflush(stderr);
    exit(1);
}

/* Calling rpc_server_use_protseq to obtain an endpoint on which to listen */

rpc_server_use_protseq("ncadg_ip_udp", MAX_CONCURRENT_CALLS, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot use protocol sequence - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* create 3 uuids....english, french, spanish */
/* one for each type */

uuid_create(&uuid_english, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot create UUID - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

uuid_create(&uuid_french, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot create UUID - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

uuid_create(&uuid_spanish, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot create UUID - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* generate seven object uuids from object strings defined in idl file */

uuid_from_string(CANADIAN_OBJECT,&canadian, &st);
uuid_from_string(BRITISH_OBJECT,&british, &st);
uuid_from_string(AMERICAN_OBJECT,&american, &st);
uuid_from_string(FRENCH_OBJECT,&french, &st);
uuid_from_string(BELGIAN_OBJECT,&belgian, &st);
uuid_from_string(MEXICAN_OBJECT,&mexican, &st);
uuid_from_string(SPANISH_OBJECT,&spanish, &st);

Figure 87 (Part 2 of 6). Greet4 Server Code — Modified for Multiple Object Types
/* Calling rpc_server_register_if to register its interface with the RPC runtime by supplying its interface specifier and EPV. Must be called three times, once for each TYPE - English, French, and Spanish */

rpc_server_register_if(greet_v1_0_s_ifspec, &uuid_english, (rpc_mgr_epv_t) &greet_v1_0_english, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot register interface - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

rpc_server_register_if(greet_v1_0_s_ifspec, &uuid_french, (rpc_mgr_epv_t) &greet_v1_0_french, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot register interface - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

rpc_server_register_if(greet_v1_0_s_ifspec, &uuid_spanish, (rpc_mgr_epv_t) &greet_v1_0_spanish, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot register interface - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

rpc_object_set_type(&canadian, &uuid_english, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot set object type - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

rpc_object_set_type(&british, &uuid_english, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot set object type - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

rpc_object_set_type(&american, &uuid_english, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot set object type - %s\n", error_text);
    fflush(stderr);
    exit(1);
}
.rpc_object_set_type(&french, &uuid_french, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, error_inq_st);
    fprintf(stderr, "Cannot set object type -\%s\n", error_text);
    fflush(stderr);
    exit(1);
}

.rpc_object_set_type(&belgian, &uuid_french, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, error_inq_st);
    fprintf(stderr, "Cannot set object type -\%s\n", error_text);
    fflush(stderr);
    exit(1);
}

.rpc_object_set_type(&mexican, &uuid_spanish, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, error_inq_st);
    fprintf(stderr, "Cannot set object type -\%s\n", error_text);
    fflush(stderr);
    exit(1);
}

.rpc_object_set_type(&spanish, &uuid_spanish, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, error_inq_st);
    fprintf(stderr, "Cannot set object type -\%s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* Calling rpc_server_inq_bindings to obtain a vector of
   binding handles that can be used to register the server's
   endpoint. The server then obtains, prints, and frees a
   string binding */

.rpc_server_inq_bindings(&bvec, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, error_inq_st);
    fprintf(stderr, "Cannot inquire bindings -\%s\n", error_text);
    fflush(stderr);
    exit(1);
}

printf("Server %s bindings:\n", entry_name);
fflush(stdout);
for (i = 0; i < bvec->count; i++) {
    rpc_binding_to_string(binding, &string_binding, &st);
    printf("%s\n", (const char *)string_binding);
    fflush(stdout);
    rpc_string_free(&string_binding, &st);
}
/* Define the array of object uids for the seven objects */

obj_uuids.count = 7;
obj_uuids.uuid[0] = &canadian;
obj_uuids.uuid[1] = &british;
obj_uuids.uuid[2] = &american;
obj_uuids.uuid[3] = &french;
obj_uuids.uuid[4] = &belgian;
obj_uuids.uuid[5] = &mexican;
obj_uuids.uuid[6] = &spanish;

/* The server endpoint is registered in the local Endpoint Map */

rpc_ep_register(greet_v1_gz'ro@ot_s_ifspec, bvec,
    (uuid_vector_t *) &obj_uuids,
    (unsigned_char_p_t) "greet version 1.0 server", &st);

if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot register to EPM: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* export the binding vector the runtime gave us to the namespace */

rpc_ns_binding_export(rpc_c_ns_syntax_dce, entry_name,
    greet_v1_gz'ro@ot_s_ifspec, bvec,
    (uuid_vector_t *) &obj_uuids, &st);

if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot export binding vector: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* To begin listening for RPC requests, the server calls
rpc_server_listen. This call is placed within the TRY of a
TRY, CATCH_ALL, ENDTRY sequence, so that if the server receives
a signal while it is listening, it can unregister its interface
and its endpoint before it exits. */

TRY {
    printf("Server %s is listening...\n", entry_name);
    fflush(stdout);
}

Figure 87 (Part 5 of 6). Greet4 Server Code — Modified for Multiple Object Types
Figure 87 (Part 6 of 6). Greet4 Server Code — Modified for Multiple Object Types

Modifying the Greet4 Client

The modifications you make to the Greet client routine in Figure 88 on page 111 are explained by the following reference keys:

1. Declare a new variable `nationality` to store the nationality code that is passed in as a parameter during client start up. Also declare a variable `obj_uuid` of type `uuid_t`.

2. Check the validity of the nationality code, and depending on the nationality code, create an object UUID from the string constant defined in the IDL file. The `if/else if` block performs this check.

3. Import the binding for the compatible Greet server using the correct object UUID based on the nationality.

4. Add a series of `if` statements to make the Greet RPC call. These `if` statements are required to pass the Greet client’s greeting to the server in the proper language.
```c
#include <stdio.h>
#include <locale.h>
#include <dce/dce_error.h>
#include "greet4.h"

int main(int argc, char* argv[]) {
    handle_t h;
    dce_error_string_t error_text;
    unsigned long st;
    int error_inq_st;
    idl_char *string_binding;
    int i, MAX_PASS;
    char reply[STR_SZ], server_name, nationality;
    rpc_ns_import_handle_t import_context;
    uuid_t obj_uuid;

    setlocale(LC_ALL, "");
    /* add extra argument to indicate nationality */

    if (argc != 4) {
        fprintf(stderr, "Usage: %s <server_entry> <passes> <nationality>\n", argv[0]);
        fflush(stderr);
        exit(1);
    }

    server_name = argv[1];
    nationality += argv[3];

    if (nationality == 'C')
        uuid_from_string (CANADIAN_OBJECT, &obj_uuid, &st);
    else if (nationality == 'B')
        uuid_from_string (BRITISH_OBJECT, &obj_uuid, &st);
    else if (nationality == 'A')
        uuid_from_string (AMERICAN_OBJECT, &obj_uuid, &st);
    else if (nationality == 'F')
        uuid_from_string (FRENCH_OBJECT, &obj_uuid, &st);
    else if (nationality == 'Q')
        uuid_from_string (BELGIAN_OBJECT, &obj_uuid, &st);
    else if (nationality == 'M')
        uuid_from_string (MEXICAN_OBJECT, &obj_uuid, &st);
    else if (nationality == 'S')
        uuid_from_string (SPANISH_OBJECT, &obj_uuid, &st);
    else {
        fprintf(stderr, "Nationality code is incorrect.");
        fflush(stderr);
        exit(1);
    }

    Figure 88 (Part 1 of 4). Greet4 Client Code — Modified for Multiple Object Types
}
```
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot convert string to object uuid: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* import compatible server bindings from the namespace */
rpc_ns_binding_import_begin(rpc_c_ns_syntax_dce,
                           server_name, greet_v1_0_c_ifspec,
                           &obj_uuid, &import_context, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot begin importing binding: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* sift through bindings and choose the first one over udp */
*****************************************************************************
For this release, RPC only works over UDP.
*****************************************************************************

while (1) {
    rpc_ns_binding_import_next(import_context, &h, &st);
    if (st == rpc_s_no_more_bindings) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot find binding over udp: %s\n", error_text);
        fflush(stderr);
        exit(1);
    }
    rpc_binding_to_string_binding(h, &string_binding, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot convert binding to string binding: %s\n", error_text);
        fflush(stderr);
        exit(1);
    }
    /* out of curiosity, print the binding */
    if (strstr(string_binding, "ncadg_ip_udp") != 0) {
        fprintf(stdout, "Client bound to server %s at %s\n",
                server_name, string_binding);
        fflush(stdout);
        rpc_string_free(&string_binding, &st);
        break;
    }
}

Figure 88 (Part 2 of 4). Greet4 Client Code — Modified for Multiple Object Types
Figure 88 (Part 3 of 4). Greet4 Client Code — Modified for Multiple Object Types
if (nationality == 'M') {
    for (i=1; i <= MAX_PASS; i++) {
        greet_rpc(h, "Ola Server!", reply);
        printf("El Greet Server dice: %s\n", reply);
        fflush(stdout);
    }
    return(0);
}

if (nationality == 'S') {
    for (i=1; i <= MAX_PASS; i++) {
        greet_rpc(h, "Ola Server!", reply);
        printf("El Greet Server dice: %s\n", reply);
        fflush(stdout);
    }
    return(0);
}

Figure 88 (Part 4 of 4). Greet4 Client Code — Modified for Multiple Object Types

Starting the Greet4 Server and Client

As noted in "Starting the Greet2 Server and Client" on page 94, you must log into DCE and ensure that DCEKERN is running and all daemons are running in your cell prior to starting the Greet server. You can start up the Greet4 server the same as you did the Greet2 server, but you should change the server entry name to /./<your_dir_name>/greet4 in this example.

After you have verified your server is in listening mode, you can start your client application. To start the Greet4 client in the Shell, use the following command:

greet4_client /./<your_dir_name>/greet4 3 S'

To start the Greet4 client in batch, use the sample JCL shown in Figure 89.

//JOBNAME JOB (ACCOUNT)...your_job_parameters
//***********************************************
//* //JCL TO STARTUP THE GREET4 CLIENT
//* //***********************************************
//GREET4CL EXEC PGM=GREET4CL,PARM='POSIX(ON)/ ./<your_dir_name>/greet4 3 S'
//SYSSOUT DD SYSOUT=* 1
//SYSPRINT DD SYSOUT=* 1
//SYSUDUMP DD SYSOUT=* 1
//STEPLIB DD DSN='USERPRFX.GREET4.LOAD'

Figure 89. Starting the Greet4 Client with Multiple Object Types

In this example, ‘S’ is passed in as an input parameter to represent Spanish. This is denoted by reference key 1. You can modify this to any of the six other nationality codes (C, A, B, F, Q, or M) and see how your client and server output changes.

Your output should appear similar to Figure 90 on page 115 when the nationality code is ‘S’.

114 DCE Application Development Guide: Introduction and Style
Client bound to server /./<your_dir_name>/greet4 at d58ab708-b3c6-11ca-891c-c9c2d4ff3b52@ncadg_ip_udp:9.21.6.97
El Greet Server dice: Buenas Dias client!
El Greet Server dice: Buenas Dias client!
El Greet Server dice: Buenas Dias client!

Figure 90. Greet4 Client Output — for Multiple Object Types
Chapter 3. Securing the Greet Application

This chapter describes two extensions to the Greet application (described in “Using the Name Service Interface” on page 86) to secure it using the DCE Security services. The examples include the following:

- Greet with name based authorization (through Authenticated RPC).
- Greet with Access Control List (ACL) based authorization.

Prior to running these applications, you should have successfully run the Greet tutorial applications contained in “Creating a Sample Application: GREET” on page 58 and Chapter 2, “Extending the Greet Application” on page 75. In addition, you should read Chapter 5, “Security” on page 163 to gain an understanding of DCE security.

For general information on using Authenticated RPC in your applications, refer to the “Security and RPC: Using Authenticated RPC” section in the Z/OS DCE Application Development Guide: Core Components.

Greet with Name-Based Authorization

This application extends the Greet application referenced in “Using the Name Service Interface” on page 86. The extensions are designated by reference keys in the following sections.

Finding the Source Code for this Example

You can find the source code for this version of the Greet example in the /usr/lpp/dce/examples/greet5 directory. The source files supplied with z/OS DCE for this example are as follows:

- greet5.idl (the IDL file)
- greet5_client.c (the client code)
- greet5_server.c (the server code)
- greet5_manager.c (the manager code)
- Makefile
- README.

If you cannot find the Greet5 example source files in the above directory, consult your systems programmer to find where they are located in your system.

The IDL file content is the same as in “Using the Name Service Interface” on page 86 except for the interface UUID. It is called greet5.idl for this example. To avoid any potential conflicts with other instances of the same Greet5 example and ensure that you are running a unique instance, you should generate your own interface UUID using the UUID Generator and replace the interface UUID in the IBM-supplied Greet5 IDL file with it.

Prior to running this example, you need to change all occurrences of your_cell_name in the Greet5 Manager example code to your local cell name.
Greet Server (Main)

Refer to Figure 91 on page 119 to see the following changes made to the Greet server code to secure the application:

1. The server defines a key table (keytab) file, designated by `KEYFILE`, where its keys are stored in encrypted form. When the Greet server program runs, the DCE runtime obtains the server’s key, (that is, its password) from this file. Normally, the DCE Administrator creates and populates the keytab file as shown in “Creating a Key Table File for the Greet Server” on page 128. The file designated by `KEYFILE` is used in the `sec_key_mgmt_get_key()` and `rpc_server_register_auth_info()` calls, so the DCE runtime knows where to obtain the key for this server application when it is registering the Greet server’s authentication information. Using a keytab file avoids having a noninteractive DCE principal from having to supply its password.

2. New declarations are made in order to use DCE security API calls.

3. The server application sets up its own login context by logging in as a user-supplied DCE principal, and discarding the login context of the user that invokes it. You must pass the DCE principal name to the server application as a start up argument, so it runs as that principal. The principal name is assigned to the variable `dce_login` in the server code. A DCE account for this principal (with permission to register its interface to DCE Host Daemon and export its bindings to the namespace) must exist for the program to run. Consult your DCE Administrator to set it up. Note that the expected server principal name in this example is `.../your_cell_name/greets`, and this is checked in the manager code.

4. The server registers its authentication information to the RPC runtime, specifying its principal name (contained in variable `dce_login`) and the authentication service (DCE shared-secret key authentication) to use when authenticating incoming calls.

#include <stdio.h>
#include <locale.h>
#include <dce/dce_error.h>
#include <dce/exc_handling.h>
#include <dce/sec_login.h>
#include <dce/keymgmt.h>
#include "greet5.h"

#define MAX_CONCURRENT_CALLS 5
#define KEYFILE "FILE:/tmp/gkeyfile"

/* Thread function definition */
static pthread_addr_t signal_thread(pthread_addr_t arg);

/* In the first part of the main function, the server assigns
the principal name (read in as an input parameter)
to the variable dce_login. This variable is used
by the server application to set up its own context.

Its CDS name (also passed as an input parameter) is assigned to entry_name.
The rpc_network_is_protseq_valid checks that its argument
specifies a protocol sequence that is supported on its host
both by the runtime library and the operating system. */

int main (int argc, char argv[])
{
    rpc_binding_vector_p_t bvec;
    unsigned long st;
    int error_inq_st;
    idl_boolean validfamily;
    idl_char string_binding;
    dce_error_string_t error_text;
    char entry_name[STR_SZ];
    int i;

    /* Declarations for Security */
    sec_login_auth_src_t auth_src;
    sec_login_handl_t login_context;
    char dce_login[STR_SZ];
    void *keydata
    boolean32 reset_passwd;

    /* Declarations for Threads */

    pthread_t thread;
    int retval;

    setlocale(LC_ALL, "");

    if (argc != 3) {
        fprintf (stderr, "Usage: %s <PRINCIPAL> <SERVER ENTRY>\n", argv[0]);
        fflush(stderr);
        exit(1);
    }

    strcpy(dce_login, argv[1]);
    strcpy(entry_name, argv[2]);

Figure 91 (Part 1 of 5). Greet5 Server Code Modified for Name-Based Security
printf("Establishing login identity with security server...
");
fflush(stdout);

sec_login_setup_identity(dce_login, sec_login_no_flags, 
           &login_context, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot setup login identity: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* Getting servers password from the authentication service */
sec_key_mgmt_get_key(rpc_c_authn_dce_secret,KEYFILE, 
           dce_login,gz'ro@ot,&keydata,&st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot get key data - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

sec_login_validate_identity(login_context, (sec_passwd_rec_t g+O#SJ) keydata, 
            &reset_passwd, &auth_src, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot validate login identity: %s\n",error_text);
    fflush(stderr);
    exit(1);
}

sec_login_set_context(login_context, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot set login context: %s\n",error_text);
    fflush(stderr);
    exit(1);
}

printf("Identity established !\n");
fflush(stdout);

/* Free the storage allocated for keydata */

sec_key_mgmt_free_key(keydata, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot free key storage: %s\n",error_text);
    exit(1);
}
/* Calling rpc_server_use_all_protseqs to tell the RPC runtime to use all supported protocol sequences */

rpc_server_use_all_protseqs(MAX_CONCURRENT_CALLS, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot establish protocol sequences: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* Calling rpc_server_register_if to register its interface with the RPC runtime by supplying its interface specifier */

rpc_server_register_if(greet_v1_0_s_ifspec, NULL, NULL, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot register interface - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* Calling rpc_server_inq_bindings to obtain a vector of binding handles that can be used to register the server's endpoint. The server then obtains, prints, and frees a string binding */

rpc_server_inq_bindings(&bvec, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot inquire bindings - %s\n", error_text);
    fflush(stdout);
    exit(1);
}

printf("Server %s bindings:\n", entry_name);
fflush(stdout);

for (i = 0; i < bvec->count; i++) {
    rpc_binding_to_string_binding(bvec->binding_h[i],
        &string_binding, &st);
    printf("%s\n", (char *)string_binding);
    fflush(stdout);
    rpc_string_free(&string_binding, &st);
}

Figure 91 (Part 3 of 5). Greet5 Server Code Modified for Name-Based Security
/* Register Authentication information with RPC runtime */

printf("Registering Authentication info...\n");
fflush(stdout);
rpc_server_register_auth_info(dce_login, rpc_c_authn_dce_secret,    
    NULL, KEYFILE, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot register authentication service: \%\n",  
            error_text);
        fflush(stderr);
        exit(1);
    }

/* The server endpoint is registered in the local Endpoint Map */

rpc_ep_register(greet_v1_gz'ro@ot_s_ifspec, bvec,     
    (uuid_vector_p_t) NULL,     
    (unsigned_char_p_t) "greet version 1.0 server", &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot register endpoint: \%\n", error_text);
        fflush(stderr);
        exit(1);
    }

/* export the binding vector the runtime gave us to the namespace */

rpc_ns_binding_export(rpc_c_ns_syntax_dce, entry_name,     
    greet_v1_gz'ro@ot_s_ifspec, bvec,     
    (uuid_vector_t *)NULL, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot export binding vector: \%\n", error_text);
        fflush(stderr);
        exit(1);
    }

/* Start thread to wait for signals */

retval = pthread_create(&thread, pthread_attr_default,     
    signal_thread, NULL);
    if (!retval) {
        pthread_detach(&thread);
    } else {
        fprintf(stderr, "Cannot create signal thread.  Server:Shutdown starting.\n");
        fflush(stdout);
    }

/* To begin listening for RPC requests, the server calls */

rpc_server_listen(MAX_CONCURRENT_CALLS, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Error: \%\n", error_text);
    }

Figure 91 (Part 4 of 5). Greet5 Server Code Modified for Name-Based Security
/* Unregister interface and endpoints before exit */

    printf("Server %s unexporting\n", entry_name);
    fflush(stdout);
    rpc_ns_binding_unexport(rpc_c_ns_syntax_dce, entry_name,
        greet_v1_0_s_ifspec, (uuid_vector_t)NULL, &st);
    printf("Unregistering endpoint \n");
    fflush(stdout);
    rpc_ep_unregister(greet_v1_0_s_ifspec, bvec,
        (uuid_vector_p_t)NULL, &st);
    printf("unregistering interface\n");
    fflush(stdout);
    rpc_server_unregister_if(greet_v1_0_s_ifspec, NULL, &st);
    printf("Purging login context\n");
    fflush(stdout);
    sec_login_purge_context(&login_context, &st);

    return(0);
}

GNUC Signal handler

static pthread_addr_t signal_thread(pthread_addr_t arg)
{
    sigset_t   signal_set;
    int        i;
    error_status_t status;

    while (true) {
        sigemptyset(&signal_set);
        sigaddset(&signal_set, SIGINT);
        sigaddset(&signal_set, SIGTERM);
        sigaddset(&signal_set, SIGHUP);
        i = sigwait(&signal_set);
        if (i == SIGINT || i == SIGTERM)
            break;
    }

    printf("Server:Shutdown starting.\n");
    rpc_mgmt_stop_server_listening(NULL, &status);
    pthread_exit(NULL);
}

Figure 91 (Part 5 of 5). Greet5 Server Code Modified for Name-Based Security

Greet Server (Manager)

Refer to Figure 92 on page 124 to see the following changes made to secure the application:

Prior to processing the client RPC request, the Greet server’s manager performs a number of checks on
the client’s binding handle to authorize the remote call. These checks are all performed in the
authorize_client() procedure denoted by 1 in the figure.

1 The authentication service specified by the client is that expected by the server (DCE shared-secret
key authentication).

2 The protection level requested by the client is that expected by the server.

3 The authorization service that expected by the server (name-based).
The server principal name specified by the client is correct. In this example, it expects the principal name `/.../your_cell_name/greets`.

The client principal name specified by the client is that expected by the server. In this example, it expects the principal name `/.../your_cell_name/greetc`.

The Greet server rejects the client's RPC if any of these checks fail, and sends a rejection message back to the client.

```c
#include <stdio.h>
#include "greet5.h"

long authorize_client(rpc_binding_handle_t);

void greet_rpc (handle_t h,
    char *client_greeting,
    char *server_reply)
{
    printf("The client says: %s\n", client_greeting);
    fflush(stdout);
    if (authorize_client (h)) {
        printf("Client is authorized.\n");
        fflush(stdout);
        strncpy(server_reply, "Hi client!", STR_SZ);
    } else {
        printf("Client is NOT authorized!\n");
        fflush(stdout);
        strncpy(server_reply, "You are not NOT authorized!", STR_SZ);
    }
}

long authorize_client (rpc_binding_handle_t bh)
{
    rpc_authz_handle_t privs;
    char *server_princ_name;
    unsigned32 protect_level, authn_svc, authz_svc, st;
    printf("Validating Client\n");
    fflush(stdout);
    rpc_binding_inq_auth_client(bh, &privs, &server_princ_name,
        &protect_level, &authn_svc, &authz_svc, &st);
    if (st != rpc_s_ok) {
        fprintf(stderr, "Cannot inquire client's authorization.\n");
        fflush(stderr);
    } else if (authn_svc != rpc_c_authn_dce_secret) {
        printf("Invalid authentication service.\n");
        fflush(stdout);
    } else if (protect_level != rpc_c_protect_level_pkt_integ) {
        printf("Invalid protection level.\n");
        fflush(stdout);
    } else if (authn_svc != rpc_c_authn_dce_secret) {
        printf("Invalid authentication service.\n");
        fflush(stdout);
    } else if (protect_level != rpc_c_protect_level_pkt_integ) {
        printf("Invalid protection level.\n");
        fflush(stdout);
    }
```
else if (authz_svc != rpc_c_authz_name) {
    printf("Invalid authorization level \n");
    fflush(stdout);
}

else if (strcmp(server_princ_name, "/.../your_cell_name/greets") != 0) {
    printf("Invalid server principal name %s \n", server_princ_name);
    fflush(stdout);
}

else if (strcmp((unsigned char_t *) privs, "/.../your_cell_name/greets") != 0) {
    printf("Invalid client principal name %s \n", (unsigned char_t *) privs);
    fflush(stdout);
}

else {
    printf("Client is valid \n");
    fflush(stdout);
    return(1);
}

return(0);

Figure 92 (Part 2 of 2). Greet5 Manager Code Modified for Name-Based Security

Greet Client

In this example, the Greet client inherits its context from the user who invokes it instead of logging into DCE with its own identity like the Greet server. Although it is possible to establish its own identity using the same calls used in the server, a client application typically inherits its context from a user rather than setting up its own identity. When you run the Greet client, you must login as the DCE principal /.../your_cell_name/greets or else the name-based authorization checking will prevent the RPC from occurring. If you want to run as a different client principal, you must change the hardcoded authorization check in the Greet Manager code. Refer to [Figure 93 on page 126] to see the following major changes in the Greet client code:

1. After importing a compatible binding from the CDS namespace, the client resolves its endpoint to obtain a full binding.

2. It then finds the Greet server's principal name to set authentication and authorization information for the Greet server binding handle.

3. This information includes the protection level, authentication service, the client's identity, and authorization service (which are all verified on the server side by the manager as part of the authorization process).

Note: In this example, the leaf entry of the Greet server's CDS name is NOT the same as the Greet server's principal name. For simplicity, you may code the server's leaf CDS entry name and principal name to be identical; however, the DCE architecture does not constrain you to keeping them the same. If your client application does not know its server's principal name, several methods are available for your client to find it. The method used is up to you. This version of Greet shows you how to find a server's principal name in the event that you do not know it at client start up, or if it is not stored somewhere for easy retrieval by the client.
```c
#include <stdio.h>
#include <locale.h>
#include <dce/dce_error.h>
#include <dce/sec_login.h>
#include "greet5.h"

int main(int argc, char*argv[]) {
  rpc_binding_handle_t h;
  dce_error_string_t error_text;
  unsigned long st;
  int error_inq_st;
  idl_char *string_binding;
  int i, MAX_PASS;
  char reply[STR_SZ], entry_name[STR_SZ];
  char *server_principal;
  rpc_ns_import_handle_t import_context;

  setlocale(LC_ALL, ":");

  if (argc != 3) {
    fprintf(stderr, "Usage: %s <SERVER NAME> <PASSES>\n", argv[0]);
    fflush(stderr);
    exit (1);
  }

  strcpy(entry_name, argv[1]);
  MAX_PASS= atoi(argv[2]);

  /* Import compatible server bindings from the namespace */

  printf("Importing a binding handle....\n");
  fflush(stdout);

  rpc_ns_binding_import_begin(rpc_c_ns_syntax_dce, entry_name,
    greet_v1_0_c_ifspec, (uuid_t *)NULL,
    &import_context, &st);

  if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot begin importing binding: %s\n", error_text);
    fflush(stderr);
    exit(1);
  }
}

Figure 93 (Part 1 of 3). Greet5 Client Code Modified for Name-Based Security
```
/ sifting through bindings and choose the first one over udp */

while (1) {

rpc_ns_binding_import_next(import_context, &h, &st);
if (st == rpc_s_no_more_bindings) {
  dce_error_inq_text(st, error_text, &error_inq_st);
  fprintf(stderr,"Cannot find binding over udp: %s\n", error_text);
  fflush(stderr);
  exit(1);
}
/* out of curiosity, print the binding */

rpc_binding_to_string_binding(h, &string_binding, &st);
if (st != error_status_ok) {
  dce_error_inq_text(st, error_text, &error_inq_st);
  fprintf(stderr,"Cannot convert binding to string binding: %s\n", error_text);
  fflush(stderr);
  exit(1);
}
printf("Client bound to server %s at %s\n",
        entry_name, string_binding);
rpc_string_free(&string_binding, &st);
}
/* end the binding import lookup loop */

rpc_ns_binding_import_done(&import_context, &st);
if (st != error_status_ok) {
  dce_error_inq_text(st, error_text, &error_inq_st);
  fprintf(stderr,"Cannot end binding import: %s\n", error_text);
  fflush(stderr);
  exit(1);
}
/* Resolve the partial binding imported from the namespace */

printf("Resolving the partially bound handle....\n");
fflush(stdout);

rpc_ep_resolve_binding (h, greet_v1_0_c_ifspec, &st);
if (st != error_status_ok) {
  dce_error_inq_text(st, error_text, &error_inq_st);
  fprintf(stderr,"Cannot resolve binding imported from CDS: %s\n", error_text);
  fflush(stderr);
  exit(1);
}

Figure 93 (Part 2 of 3). Greet5 Client Code Modified for Name-Based Security
/* Get the server principal name from the runtime */

printf("Getting the server principal name....\n");
fflush(stdout);

rpc_mgmt_inq_server_princ_name (h, rpc_c_authn_dce_secret,
    &server_principal, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot get server's principal name for server %s: %s\n",
            entry_name, error_text);
    fflush(stderr);
    exit(1);
}

printf("The server principal name is: %s\n", server_principal);
fflush(stdout);

/* Set the authorization information */

printf("Setting the Auth info....\n");
fflush(stdout);

rpc_binding_set_auth_info (h, server_principal,
    rpc_c_protect_level_pkt_integ,
    rpc_c_authn_dce_secret, NULL,
    rpc_c_authz_name, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot set security authorization for server %s: %s\n",
            entry_name, error_text);
    fflush(stderr);
    exit(1);
}

rpc_string_free(&server_principal, &st);

/* Make the remote call */

for (i=1; i <= MAX_PASS; i++) {
    greet_rpc(h, "Hello Server!", reply);
    printf("The Greet Server said: %s\n", reply);
    fflush(stdout);
}

return(0);

Figure 93 (Part 3 of 3). Greet5 Client Code Modified for Name-Based Security

Creating a Key Table File for the Greet5 Server

This application uses a local keytab file to store the keys (essentially the passwords) of the server principal. Prior to starting the Greet server application, you need to create this Key Table file using the following dcecp subcommand (DCECP in TSO/E):

dcecp -c keytab create greets -att {{storage /tmp/gkeyfile}
    {data {greets plain 1 greets}}}

This subcommand sets the password for the server principal to greets. The password in the keytab file must be the same as the password stored in the Security Registry for that principal account, or else the server will be unable to login to DCE Security. For specific information about the above command, consult the [z/OS DCE Administration Guide]. The above subcommand creates a key table file with the name gkeyfile in the directory /tmp. This name is hardcoded in the server program, so if you decide to change the name of the local keytab file, you must change it in your server program as well.
Note: Normally, there is a key management thread created in the server to periodically update the key in long running servers. For simplicity, this has not been added into this example.

Starting the Greet5 Application

You can use the same IDL file used in the version of Greet contained in "Using the Name Service Interface" on page 86. Compile the IDL file and the C source code (contained in "Finding the Source Code for this Example" on page 117), and link-edit the Greet client and server applications as before.

Prior to starting the Greet application, you must create two accounts, one for the Greet client principal, greetc, and one for the Greet server principal, greets, using the dcecp command in the Shell or DCECP in TSO/E. Set the permissions to carry out operations on the namespace for these principals using the dcecp acl command in the Shell or DCECP ACL in TSO/E. As before, the server principal must have insert permission to export to the CDS namespace directory <your_dir_name> and the insert or server permission to DCE Host Daemon. The client principal only requires read permission.

To start the Greet5 server in the Shell, in background mode, use the following commands:

```bash
export _EUV_SEC_KRB5CCNAME_FILE="/<your_home_dir>/krb5ccname1"
greet5_server greets /.:/<your_dir_name>/greet5 &
```

To start up the Greet5 server in batch, use the example JCL shown in Figure 94.

```plaintext
//JOBNAME JOB (ACCOUNT),...your_job_parameters
//******************************************************************************
//*
//* JCL TO STARTUP THE GREET5 SERVER
//*
//******************************************************************************
//GREET5SR EXEC PGM=GREET5SR,PARM='POSIX(ON),
//ENVAR(''_EUV_SEC_KRB5CCNAME_FILE=/<your_home_dir>/krb5ccname1''),
//  / greets /.:/<your_dir_name>/greet5'
//SYSOUT DD SYSOUT=g+O#SJ
//SYSPRINT DD SYSOUT=g+O#SJ
//CEEDUMP DD SYSOUT=g+O#SJ
//STEPLIB DD DSN=USERPRFX.GREET5.LOAD,DISP=SHR
```

Figure 94. Starting the Greet5 Server in Batch

After ensuring the Greet5 server is in listening mode, you need to log in to DCE as principal greetc prior to running the Greet5 client. The Greet5 client inherits your login context.

To run the Greet5 client in the Shell, use the following commands:

```bash
export _EUV_SEC_KRB5CCNAME_FILE="/<your_home_dir>/krb5ccname"
greet5_client /.:/<your_dir_name>/greet5 <passes>
```

To run the Greet5 client in batch, use the example JCL shown in Figure 95 on page 130.
Press any key to continue...

Figure 95. Starting the Greet5 Client in Batch

Greet5 Server Output

You will see the following output when you run the Greet5 server program:

- If the Greet5 server accepts the client’s credentials and allows the remote call:

  Establishing login identity with security server...
  Identity established !
  Server ./<your_dir_name>/greet5 bindings:
  nca0_ip_udp:9.21.22.97 [1312]
  nca0_ip_tcp:9.21.22.97 [1668]
  Registering Auth info...
  Server ./<your_dir_name>/greet5 is listening...
  The client says: Hello Server !
  Validating Client
  Client appears valid !
  Client is authorized.
  The Greet server said: Hi Client!

Figure 96. Greet5 Server Output — If Client Is Accepted

- If the Greet5 server rejects the client’s credentials and disallows the remote call:

  The client says: Hello Server !
  Validating Client
  Invalid client principal name /.../your_cell_name/Invalid_Principal.
  Client is NOT authorized !

Figure 97. Greet5 Server Output — If Client Rejected

- If the Greet5 server is stopped:

  Server ./<your_dir_name>/greet5 unexporting
  Unregistering endpoint
  Unregistering interface
  Purging login context

Figure 98. Greet5 Server Output — When Server Is Stopped
Greet5 Client Output

You will see the following Greet5 client output:

- On a successful remote call to the Greet5 server:

  Importing a binding handle....
  Client bound to server /.../<your_dir_name>/greet5 at ncadg_ip_udp:9.21.22.97
  Resolving the partially bound handle....
  Getting the server principal name....
  The server principal name is: /.../your_cell_name/greets
  Setting the Auth info....
  The Greet Server said: Hi client!

  Figure 99. Greet5 Client Output — On Successful Remote Call

- On an unsuccessful remote call to the Greet5 server:

  Importing a binding handle....
  Client bound to server /.../<your_dir_name>/greet5 at ncadg_ip_udp:9.21.22.97
  Resolving the partially bound handle....
  Getting the server principal name....
  The server principal name is: /.../your_cell_name/greets
  Setting the Auth info....
  The Greet Server said: You are NOT authorized!

  Figure 100. Greet5 Client Output — On Unsuccessful Remote Call

Greet with EPAC-based Authorization

This example adds a pre-OSF DCE 1.1 level (pre-OS/390® DCE) Access Control List (ACL) manager to the last Greet example. See the example discussed in Appendix A, “A Sample Application” on page 271 for an illustration of how to write a DCE 1.1 (z/OS DCE) ACL manager. The ACL manager contains procedures that control a client’s access to the Greet server’s manager routines. An ACL is a list of access control entries that are used in DCE to protect objects by specifying permissions granted to DCE principals and groups. The DCE dcecp tool is used to maintain the ACLs for this Greet6 example.

Note that this is a very simple ACL manager example. It only checks group and user entry types for correct permissions. Also, any principal can change permissions using dcecp.

Figure 101 on page 132 represents the traditional Greet application without security and the path that the unauthenticated RPC call (greet_rpc) takes from the client code that invokes it, to the Greet server manager routine.
Figure 102 on page 133 represents the Greet6 application with an ACL manager that protects the Greet manager routine from unauthorized client access. In this example, there are two new user-written modules added that were not required in previous examples:

- The ACL Manager interface.

This module contains the server side calls prefixed by sec_acl_mgr. The Greet6 server application uses these functions to make DCE-conformant runtime authorization decisions. The module consists of the following routines that are called from the Greet server program.

- **sec_acl_mgr_configure()**
  Configures the ACL database file.
- **sec_acl_mgr_get_access()**
  Finds the total permissions a principal has on the object. If the principal has a user entry and a group entry in the ACL, the permissions are combined. Typically, permissions are not combined if a principal matches both a user and a group entry. Permissions are typically combined only if the principal matches multiple group entries.
- **sec_acl_mgr_get_manager_types()**
  Returns all of the available manager types. Because the Greet6 example consists of only a single manager type, that is all that is returned.
- **sec_acl_mgr_is_authorized()**
  Checks to see if a principal is authorized. It checks user and group entries in the local cell.
- **sec_acl_mgr_lookup()**
  Retrieves the ACL from the ACL database.
- **sec_acl_mgr_replace()**
  Writes the ACL to the ACL database file.

- The rdacl (ACL Network Interface) Manager.

This module contains the server side API calls prefixed by rdacl. These calls enable the Greet server to communicate with sec_acl-based clients such as the acl_edit tool.

- **rdacl_lookup()**
  Returns a pointer to an ACL for an object. It loads a copy of an object’s ACL corresponding to the specified manager type into memory.
- **rdacl_replace()**
  Replaces an ACL pointed to by an input handle. As ACLs are immutable, you use this call to replace an ACL after you have modified it.
**rdacl_test_access()**
Tests access to an object by determining if a specified ACL contains entries that grant privileges to the calling process that match desired privileges.

**rdacl_test_access_on_behalf()**
Tests access to an object on behalf of a process other than the calling process.

**rdacl_get_manager_types()**
Returns a list of the types of ACLs protecting an object. ACL editors and browsers use this operation to determine the ACL manager types that a particular reference monitor is using to protect a selected identity.

**rdacl_get_printstring()**
Returns an array of printable representations or *printstrings* for each permission bit or the combination of permission bits a specified ACL manager supports.

**rdacl_get_access()**
Determines the complete extent of access to a specified object by the calling process by reading a privilege attribute certificate.

---

**Figure 102. Greet6 Application with ACL Manager**

To simplify this example, the Greet6 ACL manager handles only local cell principals and groups. There is no checking of foreign owners or groups, and no facility to store them.
Finding the Source Code for this Example

You can find the source code for this version of the Greet example in the `/usr/lpp/dce/examples/greet6` directory. The source files supplied with z/OS DCE for this example are as follows:

- `greet6.idl` (Greet6 IDL file)
- `greet6_client.c` (Greet6 client code)
- `greet6_server.c` (Greet6 server code)
- `greet6_manager.c` (Greet6 manager code)
- `greet6_secacl.c` (Greet6 ACL manager interface code)
- `greet6_rdacl.c` (Greet6 ACL network interface code)
- Makefile
- README.

If you cannot find the Greet6 example source files in the above directory, consult your systems programmer to find where they are located in your system.

In this example you must compile two IDL files to generate the necessary stubs and header files:

- `greet6.idl` (the IDL file for the Greet application)
- `rdaclifv0.idl` (the IDL file for the DCE ACL network interface).

To avoid any potential conflicts with other instances of the same Greet6 example and ensure that you are running a unique instance, you should generate your own interface UUID using the UUID Generator and replace the interface UUID in the IBM-supplied Greet6 IDL file with it.

You can find the code for this example listed in Appendix C, “Greet6 ACL Manager Example” on page 303. The code is not listed in this section because it is quite lengthy.

Changing Name-Based Authorization to EPAC-Based: Note a change in the `authorize_client()` routine contained in the Greet6 Manager code which changes the authorization checking from name-based authorization checking in the previous example, to EPAC-based authorization checking. Essentially, the name-based check in the previous example (see the code clip in Figure 103 on page 135) is replaced by the code in Figure 104 on page 136.

In the name-based example, the variable `privs` contains the client’s principal name that is returned from the `rpc_binding_inq_auth_client()` call, and used in the authorization check. The Greet5 client in the name-based example specified name-based authorization by setting `authz_svc` to `rpc_c_authz_name` in the `rpc_binding_set_auth_info()` call.
else if (strcmp((unsigned_char_t *) privs, "/.../your_cell_name/greetc") != 0) {
    printf("Invalid client principal name %s.\n",
           (unsigned_char_t *) privs);
    fflush(stdout);
}

else {
    printf("Client is valid !\n");
    fflush(stdout);
    return(1);
}


Figure 103. Greet5 Example — Name-Based Authorization

In the EPAC-based example, privs points to a data structure that contains the client’s EPAC, which is returned from the rpc_binding_inq_auth_client() call. The client in the Greet6 EPAC-based example specifies DCE (or EPAC-based) authorization by setting authz_svc to rpc_c_authz_dce in the rpc_binding_set_auth_info() call.

Figure 104 on page 136 shows the authorization checking after the client is authenticated. The value returned from the sec_acl_mgr_is_authorized() routine determines whether the client is authorized or not.
printf("Client is valid!\n");  
fflush(stdout);  
greetprivils = (sec_id_pac_t *) privs;  
printf("Authenticated --> %d \n", greetprivils->authenticated);  
fflush(stdout);  

greetprivils->principal.uuid, &str_principal,&st);  
greetprivils->group.uuid, &str_group,&st);  
greetprivils->realm.uuid, &str_realm,&st);  
printf("Principal uuid -> %s
",str_principal);  
fflush(stdout);  
printf("Group uuid -> %s
",str_group);  
fflush(stdout);  
printf("Realm uuid -> %s
",str_realm);  
fflush(stdout);  
/* Retrieving the ACL UUID */  
sec_acl_mgr_get_manager_types(NULL,NULL,sec_acl_type_object,1,&size_used,  
&num_types,manager_types,&st);  
printf(stdout, "Manager uuid %s
",str_uuid);  
fflush(stdout);  
rpc_ss_enable_allocate();  
x = sec_acl_mgr_is_authorized(NULL,required_access,greetprivils,NULL,  
&manager_types[0]), NULL,NULL,&st);  
rpc_ss_disable_allocate();  
return(x);  
}

Figure 104. Greet6 Example — EPAC Based Authorization

Building the Greet6 Example

Prior to compiling this example, you need to do the following tasks for your environment:

- Change all occurrences of your_dir_name in the Greet6 client and server example code to your local namespace directory name where the Greet server and ACL manager bindings are exported.
- Change all occurrences of your_cell_name in the Greet6 manager code.

In the Shell: Use the following makefile to compile and link-edit the Greet6 application:
IF = greet6

IDL = /bin/idl
IDL_FLAGS = -no_cpp -keep c_source
RDACLIF_PATH = /usr/include/dce
CFLAGS = -DMVS -D_DCE_THREADS -D_OPEN_SYS -Wgz'ro@ot,DLL
LIBS = -l dce /usr/lib/EUVPDLL.x

FROMIDL = $(IF).h $(IF)_cstub.c $(IF)_sstub.c rdaclif.h rdaclif_sstub.c
COBJ = $(IF)_client.o $(IF)_cstub.o
SOBJ = $(IF)_server.o $(IF)_sstub.o $(IF)_manager.o $(IF)_secacl.o $(IF)_rdacl.o rdaclif_sstub.o

default: $(IF)_client $(IF)_server
$(IF)_client: $(COBJ)
c89 -o $(IF)_client $(COBJ) $(LIBS)
$(IF)_server: $(SOBJ)
c89 -o $(IF)_server $(SOBJ) $(LIBS)
client.o server.o manager.o: $(IF).h
$(COBJ): $(IF).h
$(SOBJ): $(IF).h
$(FROMIDL): $(IDL) $(IF).idl $(RDACLIF_PATH)/rdaclif.idl
$(IDL) $(IF).idl $(IDL_FLAGS)
$(IDL): $(RDACLIF_PATH)/rdaclif.idl $(IDL_FLAGS) -client none

clean:
 rm -f $(FROMIDL) *.o

Figure 105. Sample Makefile to Build the Greet6 Application

In Batch: After you have allocated your data sets using the TSO/E IDLALLOC command, copy the HFS source files into these data sets using OGET. Rename the source files to the suggested PDS naming convention as presented in "Step 1. Creating Files for Your Application" on page 30. You also have to rename the greet6_secacl.c and greet6_rdacl.c files as PDS members. In this example, they are renamed to GREET6SA and GREET6DA respectively. Additionally, you need to copy and rename the rdaclifv0.idl file found in /usr/include/dce to an IDL member name that is no more than 6 characters. In this example, it is renamed to DACLIF. As a result, you have to change the #include rdaclifv0.h to #include daclif.h in the Greet6 server module, otherwise you will receive an error when you compile this application. In addition, you must copy rdaclibas.idl into the GREET.IDL PDS.

After you have copied the example code to your GREET6 application data sets and compiled the two IDL files, you can use the following example JCL to compile the GREET6 source code:
/* JOBNAME   JOB(ACCOUNT),...your_job_parameters */
/* JCL to compile the client, server, and stubs code */
/* DCEPFX - for DCE header files */
/* LNGPFX - for LE header files */
/* //g+O#SJ EXEC EDCC,INFILE='USERPRFX.C(GREET6CL)', */
/* CPARM='LO,NOMAR,NOSEQ,DLL,RENT,DEFINE(_OPEN_SYS,_DCE_THREADS,MVS)', */
/* OUTFILE='USERPRFX.OBJ(GREET6CL),DISP=SHR', */
/* LNGPFX='CEE',DCEPFX='DCE' */
/* USERLIB DD DSN=USERPRFX.GREET6.H,DISP=SHR */
/* SYSLIB DD DSN=DCEPFX..SEUVHDR,DISP=SHR */
/* DD DSN=NGPFX..SCEEH.H,DISP=SHR */
/* DD DSN=NGPFX..SCEEH.SYS.H,DISP=SHR */
/* //g+O#SJ EXEC EDCC,INFILE='USERPRFX.C(GREET6CS)', */
/* CPARM='LO,NOMAR,NOSEQ,DLL,RENT,DEFINE(_OPEN_SYS,_DCE_THREADS,MVS)', */
/* OUTFILE='USERPRFX.OBJ(GREET6CS),DISP=SHR', */
/* LNGPFX='CEE',DCEPFX='DCE' */
/* USERLIB DD DSN=USERPRFX.GREET6.H,DISP=SHR */
/* SYSLIB DD DSN=DCEPFX..SEUVHDR,DISP=SHR */
/* DD DSN=NGPFX..SCEEH.H,DISP=SHR */
/* DD DSN=NGPFX..SCEEH.SYS.H,DISP=SHR */
/* //g+O#SJ EXEC EDCC,INFILE='USERPRFX.C(GREET6SR)', */
/* CPARM='LO,NOMAR,NOSEQ,DLL,RENT,DEFINE(_OPEN_SYS,_DCE_THREADS,MVS)', */
/* OUTFILE='USERPRFX.OBJ(GREET6SR),DISP=SHR', */
/* LNGPFX='CEE',DCEPFX='DCE' */
/* USERLIB DD DSN=USERPRFX.GREET6.H,DISP=SHR */
/* SYSLIB DD DSN=DCEPFX..SEUVHDR,DISP=SHR */
/* DD DSN=NGPFX..SCEEH.H,DISP=SHR */
/* DD DSN=NGPFX..SCEEH.SYS.H,DISP=SHR */

Figure 106 (Part 1 of 2): Sample JCL to Compile the Greet6 ACL Manager Application
Figure 106 (Part 2 of 2). Sample JCL to Compile the Greet6 ACL Manager Application

Use the example JCL in the following figure to link-edit the GREET6 client application.
Figure 107. Sample JCL to Link-Edit the Greet6 Client

Use the example JCL in the following figure to link-edit the GREET6 server application.

Figure 108. Sample JCL to Link-Edit the Greet6 Server
Starting the Greet6 Server

The Greet6 ACL manager controls the Greet6 client's access to run the `greet_rpc()` function. When the server starts up, the Greet6 server binding information is exported to the `./:<your_dir_name>/greet6` entry in the namespace, and the ACL binding information is exported to the `./:<your_dir_name>/greet_acl` entry. Notice that this example also registers two interfaces with the DCE runtime:

- `greet_v1.0_s_ifspec`. This interface is the same as in previous versions of the Greet application, and handles the actual Greet RPC.
- `rdaclif_v0.0_s_ifspec`. This interface is required to perform the ACL management using the `dcecp` tool.

To start the Greet6 server in the Shell, in background mode, use the following command:

```
export _EUV_SEC_KRB5CCNAME_FILE='/<your_home_dir>/krb5ccname1'
greet6_server greets greets &
```

To start up the Greet6 server in batch, use the following example JCL:

```
//JOBNAME JOB (ACCOUNT),...your_job_parameters
/***********----------------------------------------------------------------------------
/*
/* JCL TO STARTUP THE GREET6 SERVER
/*
/***********----------------------------------------------------------------------------
//GREET6SR EXEC PGM=GREET6SR,PARM='POSIX(ON),
 'ENVAR(''_EUV_SEC_KRB5CCNAME_FILE=/<your_home_dir>/','krb5ccname1'')/ greets greets')
//SYSOUT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//CEEDUMP DD SYSOUT=*
//STEPLIB DD DSN=USERPRFX.GREET6.LOAD,DISP=SHR
```

Figure 109. Sample JCL to Start the Greet6 Server

Figure 110 shows the output you should see for the Greet server initialization process until it is in listening. You can trace through the steps in the initialization process.

```
Establishing login identity with security server...  
Identity established !  
Initializing ACL database  
Inside sec_acl_mgr_configure  
Getting ACL manager types  
Inside sec_acl_mgr_get_manager_types  
Leaving sec_acl_mgr_get_manager_types  
Setting Protocol sequence...  
Registering Greet interface...  
Registering ACL interface...  
Server Greet bindings:  
ncadg_ip_udp:9.21.22.97 [1373]  
ncacn_ip_tcp:9.21.22.97 [1667]  
Registering Greet Server Auth info...  
Registering Greet interface with EPM...  
Registering ACL interface with EPM...  
Exporting server bindings to namespace...  
Exporting ACL bindings to namespace...  
Greet server is listening...
```

Figure 110. Greet6 Server Initialization Output
Updating the ACL Database File
The initial content of the Greet6 ACL database file is shown in Figure 111.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>greet</td>
<td>1</td>
</tr>
<tr>
<td>09A31250-16C8-1BB8-9B7D-C9C2D4FF0027</td>
<td>2</td>
</tr>
<tr>
<td>00668186-B089-1855-A97A-10005AA89D46</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 111. Greet6 ACL Database file (After initialization)

1. The first entry is a temporary character string, greet, that is inserted by the server application. This gets updated to your local cell name when you use the -cell /.: option on the dcecp acl modify command.

2. This is the cell UUID.

3. This is the ACL manager UUID. The Greet6 ACL manager program uses this UUID to verify the correct ACL manager type.

4. This is the number of ACLs contained in the ACL database file. Initially, this is set to 0 (zero); however, as you add or delete ACLs from this file using dcecp, this number gets updated.

You can update the ACLs for this application only after the server is initialized and running, that is, the server is in listening mode. To update the ACL database file for this example, use the dcecp tool.

When the Greet server program initializes the ACL, it inserts a temporary cell name. In this case, set the defaults to their correct identities by using the cell /.: option from within the dcecp command.

To update permissions for principals to modify the Greet6 ACL database file, use the acl modify subcommand within dcecp. For example, to update the permissions for the greet server principal, greets, so it has read, write, delete permission to the ACL database file, use the following dcecp command:

dcecp -c acl modify /.:/<your_dir_name>/greet_acl -add {user greets rwd}

In this example, the greet client principal, greetc, requires a special 'g' permission created for this ACL manager example. The permissions are hardcoded in the greet6_secacl.c module as follows:
These permissions are from the dacl_manager example. A new permission "g" has been added for this Greet example.

```c
static sec_acl_printstring_t hardcoded_printstrings[] = {
    { "g", "greet", sec_acl_perm_unused },
    { "c", "control", sec_acl_perm_owner   },
    { "r", "read",  sec_acl_perm_read     },
    { "w", "write", sec_acl_perm_write    },
    { "x", "execute", sec_acl_perm_execute },
    { "i", "insert", sec_acl_perm_insert  },
    { "d", "delete", sec_acl_perm_delete  },
    { "t", "test",  sec_acl_perm_test     }
};
```

Figure 112. Greet6 ACL Manager Permissions

To update the permission for the `greet` principal, so it can run the Greet server RPC call, use the following `dcecp` command:

```
dcecp -c acl modify /.:/<your_dir_name>/greet_acl -add {user greet g}
```

You can list the permissions available using the following command:

```
dcecp -c acl permissions /.:/<your_dir_name>/greet_acl
```

You will see the new information being written to the ACL database on the server side, similar to that shown in Figure 113 on page 144. The output shown is from the routines contained in the `greet6_rdacl.c` and `greet6_secacl.c` modules after `dcecp` is invoked, and continues from the 'Greet server is listening...' status message.
* * *

Server Greet is listening...
Inside rdacl_get_mgr_types_semantics...
Inside sec_acl_mgr_get_manager_types
Leaving sec_acl_mgr_get_manager_types
Leaving rdacl_get_mgr_types_semantics.
Inside rdacl_get_printstring...
Inside sec_acl_mgr_get_printstring...
Leaving sec_acl_mgr_get_printstring.
Leaving rdacl_get_printstring
Inside rdacl_lookup...
This is the component name
Inside sec_acl_mgr_lookup...
Opening GREETACL
Getting GREETACL database info
1 /.../your_cell_name
2 09A31250-16C8-1BB8-9B7D-C9C2D4FF0027
3 00668186-BD09-1B55-A97A-10005AA89046
4 0
Users...
Leaving sec_acl_mgr_lookup...
Leaving rdacl_lookup
Inside sec_acl_mgr_replace
/.../your_cell_name 09A31250-16C8-1BB8-9B7D-C9C2D4FF0027 00668186-BD09-1B55-A97A-10005AA89046 2
128 3 greetc 00000109-0246-2C27-B500-C9C2D4FF005A
35 3 greets 00000109-023F-2C27-B500-C9C2D4FF005A
Leaving sec_acl_mgr_replace
Leaving rdacl_replace

Figure 113. Greet6 Server Output — After ACLs Are Updated

The ACL manager database file, GREETACL, is accessed solely by the manager_type UUID. The routine get_database_name() maps the UUID to the file name. Everything is handled by UUID.

Starting the Greet6 Client

To start the Greet6 client in the Shell, use the following command:

```bash
export _EUV_SEC_KRB5CCNAME_FILE="/<your_home_dir>/krb5ccname"
greet6_client greetc greetc greets 1
```

To start up the Greet6 client in batch, use the following example JCL:

```jjj
//JOBNAME JOB (ACCOUNT),...your_job_parameters
//******************************************************************************
//*/
//* JCL TO STARTUP THE GREET6 CLIENT
//*/
//******************************************************************************
//GREET6CL EXEC PGM=GREET6CL,PARM='greetc greetc greets 4'
//SYSOUT DD SYSOUT=
//SYSPRINT DD SYSOUT=
//CEEDUMP DD SYSOUT=
//STEPLIB DD DSN=USERPRFX.GREET6.LOAD,DSP=SHR
```

Figure 114. Starting the Greet6 Client

If the required client permission ("g") is in place, the RPC will proceed. Figure 115 on page 145 shows the Greet server output when the Greet ACL Manager allows the Greet RPC call to proceed.
Leaving rdacl_replace
The client says: Hello Server!
Validating Client
Client is valid!
Authenticated --> 1
Principal uuid -> 0000010A-0246-2C27-B500-C9C2D4FF005A
Group uuid -> 00000072-0267-2C27-B501-C9C2D4FF005A
Realm uuid -> 09A31250-16C8-1BB8-987D-C9C2D4FF0027
Inside sec_acl_mgr_get_manager_types
Leaving sec_acl_mgr_get_manager_types
Manager uuid 00668186-BD89-1B55-A97A-10005AA89D46
Inside sec_acl_mgr_is_authorized...
Inside sec_acl_mgr_lookup...
Opening GREETACL
Getting GREETACL database info
1 /.../your_cell_name
2 09A31250-16C8-1BB8-987D-C9C2D4FF0027
3 00668186-BD89-1B55-A97A-10005AA89D46
4 2
Users...
128 3 greetc 0000010A-0246-2C27-B500-C9C2D4FF005A
35 3 greets 00000109-023F-2C27-B500-C9C2D4FF005A
Leaving sec_acl_mgr_lookup...
Validating user uuid
Inside grant_access...
Leaving grant_access...
Leaving sec_acl_mgr_is_authorized...
Client is authorized

Figure 115. Greet6 Server Output on a Successful RPC

If the required client permissions are not in place, the RPC call will not proceed. Figure 116 on page 146 shows the Greet6 server output when the RPC is not allowed by the Greet6 ACL Manager.
Leaving rdacl_replace
The client says: Hello Server!
Authenticated --> 1
Principal uuid --> 000000A7-EDC2-2B6F-AB00-C9C2D4FF002C
Group uuid --> 00000071-EE25-2B6F-AB01-C9C2D4FF002C
Realm uuid --> 295C6100-829E-1B38-80EC-C9C2D4FF00A9
Inside sec_acl_mgr_get_manager_types
Leaving sec_acl_mgr_get_manager_types
Manager uuid 00668186-B089-1855-A97A-10005AA89D46
Inside sec_acl_mgr_is_authorized...
Inside sec_acl_mgr_lookup...
Opening GREETACL
Getting GREETACL database info
1 /.../your_cell_name
2 295C6100-829E-1B38-80EC-C9C2D4FF00A9
3 00668186-B089-1855-A97A-10005AA89D46
4 2
Users...
163 0 greets 000000A6-EDB3-2B6F-AB00-C9C2D4FF002C
Test greets
163 0 greetc 000000A7-EDC2-2B6F-AB00-C9C2D4FF002C
Test greetc
Leaving sec_acl_mgr_lookup...
Leaving sec_acl_mgr_is_authorized...
Client is NOT authorized!

Figure 116. Greet6 Server Output on an Unsuccessful RPC

The client output on an authorized RPC is shown in Figure 117.

Establishing Identity with Security Server...
Identity established!
Importing a binding handle...
Setting Auth info...
Making RPC call...
The Greet Server said: Hi Client!
Purging Login context

Figure 117. Greet6 Client Output on Authenticated RPC

The contents of the ACL database file after the two principals greets and greetc have been added are shown in Figure 118.

/.../your_cell_name
09A31250-16C8-1888-9B7D-C9C2D4FF0027
00668186-B089-1855-A97A-10005AA89D46
2
128 3 greetc 0000010A-0246-2C27-B500-C9C2D4FF005A
35 3 greets 00000109-023F-2C27-B500-C9C2D4FF005A

Figure 118. Greet6 ACL Database File (After Updates)

Notice that the number of ACLs has been updated to 2 after the ACLs for the two user principals are added. This is followed by the ACLs themselves. Scanning horizontally across each ACL record, note that it consists of:
1. A number representing the permissions for that ACL. This is set to 128 for the first ACL record in the above example.

2. A number representing the ACL entry type, either user or group in this simple ACL Manager example. The number in this example is 3 for both ACLs as they have been set to entry type of user.

3. The user principal name.

4. The user principal UUID.
Chapter 4. Threads

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

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

These issues are covered in "Thread Use Policy" on page 150.

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

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

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

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

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

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

As a result, such rules have in some sense the flavor of policy recommendations: they are a set of disciplines for avoiding certain classes of problems which threads programmers can assume to exist, in general, even though they might not arise in specific cases. Because of this, and because these rules may be unfamiliar to many programmers, it seems wise to repeat them in summary form in this policy guide. Moreover, because OSF DCE client and server applications are implicitly multithreaded, even when the application itself makes no thread related calls, it is also important to identify when application code must be thread safe. These issues are covered in "Thread Safety" on page 152.

The remaining sections of this chapter cover a variety of specific policy and usage issues relating to OSF DCE threads. Thread handles and thread-private data are discussed in Chapter 4, "Threads." Cancels and signals introduce a number of specific semantic issues that applications must be aware of when
programming in a multithreaded environment. These are covered in "Canceling Threads" on page 155 and "Signals" on page 159, respectively. Finally, OSF DCE introduces the concept of an RPC thread. This is intended to extend the semantics of a local thread of execution across two address spaces in the course of an RPC. However, the extension is not entirely transparent, and applications need to be aware of the semantic peculiarities of RPC threads. These are covered in "RPC Threads" on page 162.

### Important Note to Readers

This chapter is intended as a general guide to thread use as it pertains to all OSF DCE threads implementations. Differences in the z/OS DCE implementation of threads from the general case described are indicated by special notes in the text.

### Thread Use Policy

Thread use policy questions arise in two ways:

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

### To Thread or Not to Thread

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

**Note:** This is not the case with z/OS DCE threads, which supports true parallel processing.

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

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

In summary, the developer must consider the following questions in order to decide whether an application will benefit from multithreading:
• Are the threaded operations likely to block, for example, because they make blocking I/O calls or
RPCs? If so, then multithreading is likely to be beneficial in any implementation or hardware
environment.

• Can the underlying hardware and RPC implementation support threads on more than one processor
within a single process? If not, then multithreading cannot achieve real parallelism for processor
intensive operations. The general OSF DCE user-space threads implementation restricts all threads of
a single process to contend for a single processor and so cannot provide real parallelism for processor
intensive operations.

• Even if the answer to both of the first two questions is “yes,” will the use of a time-slicing thread
scheduling policy permit fairer distribution of server resources among contending clients? If so, then
server manager multithreading may be beneficial.

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

How Many Threads?

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

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

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

Scheduling Policies

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

Note: In z/OS DCE, MVS sets its own thread scheduling policy. Thread-creating applications, even DCE
servers, cannot change it.
Thread Safety

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

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

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

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

*Note:* `malloc()` and the rest of the C/C++ runtime routines are thread-safe.

These thread safety issues arise in two contexts for OSF DCE applications.

1. Even when application code is not itself multithreaded (for example, client code that does not make any explicit `pthread` API calls), both client and server applications are still multithreaded as a result of threads created by the RPC runtime. While such single-threaded application code need not itself be reentrant, it must still avoid blocking the entire process, and it must take care that any library routines that it calls, which may also be called by runtime-created threads, are reentrant.
2. When application code is itself multithreaded (which is the default for server manager code), it must, in addition to obeying the rules in 1), also be reentrant: all access to shared objects must be protected by locks.

Obviously, providing for the second condition in explicitly multithreaded code is the application's responsibility. The `pthread` API provides a set of facilities that can be used for this purpose. To provide for the first condition, which applies to all application code, C/C++ provides thread-safe routines.

Thread Rules

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

- Access to all shared objects should be protected by the appropriate synchronization mechanisms. The `pthread` global lock is not appropriate for such synchronization.
- Mutexes should only be used to protect resources held for a short period of time. In particular, note that `pthread_mutex_lock()` is *not* a cancelation point. Resources needing to be held exclusively for a long time should be protected by condition variables rather than mutexes, as this will not inhibit cancelability (see *Cancellation Points* on page 156).
- A shared object should be protected by only one mutex.
Be sure to use the available thread-safe library calls. These may be available as wrapped routines, via the `pthread.h` header file, or your implementation may supply reentrant libraries which must be linked with OSF DCE applications.

Avoid nonwrapped process-blocking system calls, such as `wait()`.

When threads need to acquire more than one mutex at a time, create a locking sequence and require that all threads follow the sequence.

Do not make any assumptions about the atomicity of operations, as these are unlikely to be portable.

In general, to avoid priority inversion, when three or more threads of different priorities access a lock, associate a priority with the lock and force any thread to raise its priority to the lock priority before acquiring the lock. Note that the default scheduling policy (`SCHED_OTHER`) mitigates the effects of priority inversion by giving low-priority threads a chance to execute (and thus release held locks) even when higher-priority threads are eligible to run.

Note: This is also the case in z/OS DCE threads, as MVS scheduling policy also ensures that low-priority threads are eligible to run.

You may be able to use the global locking call `pthread_lock_global_np` when calling into libraries not known to be thread safe.

Use the `atfork()` routine to keep the state of mutexes consistent across calls to `fork()`. Note, however, that this routine is not considered to be portable. You should try to create threads rather than processes whenever possible.

Note: `atfork` is not supported by z/OS DCE threads.

Call `pthread_cond_wait()` from within a predicate loop, as in:

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

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

Note: This is not applicable to z/OS DCE threads, who cannot change their scheduling attributes.

Also see "Canceling Threads" on page 155 and "Signals" on page 159 for specific guidelines relating to cancels and signals.

---

**Threads Programming Topics**

The following subsections discuss these aspects of multithreaded OSF DCE application development:

- Thread Handles and their Use
- Storage for Thread Specific Data
- Canceling Threads
- Signals
Thread Handles

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

Storage for Thread Specific Data

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

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

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

pthread_key_t key;

main()
{

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

    status = pthread_keycreate(&key, (pthread_destructor_t) destroy);

    /* Start some threads... */

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

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

    void write_global(mytype value)
    {
        mytype *global_var;

        global_var = (mytype*) malloc(sizeof(mytype));
        pthread_setspecific(key, (pthread_addr_t)global_var);
        *global_var = value;
    }
```
mytype read_global()
{
    mytype *global_var;

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

Canceling Threads

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

- **disabled**
  - No cancelation takes place.
- **deferred**
  - Cancelation is deferred to cancelation points.
- **asynchronous**
  - Cancelation may occur at any time.

The default action for OSF DCE threads on cancelation is that the thread calls any cancel cleanup routines that have been established and then terminates. In OSF DCE threads a canceled thread receives a cancel as an exception, so a thread may establish a nondefault action by providing an exception handler. However, this behavior is not recommended for two reasons. First, the exception handling mechanism is not itself portable. Second, the cancel mechanism is intended to provide for orderly thread termination. It is not designed as a generalized thread synchronization mechanism. (There is, for example, only one kind of cancel.) Threads should use condition variables for this purpose. (For the same reason, the use of `pthread_signal_to_cancel_np()` is not recommended, as explained in “Cancel Rules Summary” on page 158.)

**Cancelability State:** A thread’s cancelability state is determined by the combination of two substates: *general cancelability* and *asynchronous cancelability*. These substates can be set to either `CANCEL_ON` or `CANCEL_OFF` by calls to the routines `pthread_setcancel()` and `pthread_setasynccancel()` respectively. A thread’s cancelability state is determined by its general and asynchronous cancelability substates as follows:
One awkwardness introduced by this mechanism for setting cancelability state is that threads cannot easily determine their current cancelability state, although `pthread_setcancel()` and `pthread_setasynccancel()` return the previous substates. When a thread is created, the default cancelability state is `deferred` (general cancelability set to `CANCEL_ON`, asynchronous cancelability set to `CANCEL_OFF`). A thread that needs to discover its current cancelability state should explicitly maintain this state in some place where it can be easily queried.

**Cancelation Points:** Cancelation can occur at different points during a thread's execution, depending on the cancelability state.

If the cancelability state is `asynchronous`, then any point can be a cancelation point; that is, the thread may be canceled at any time.

If the cancelability state is `deferred` then cancelation may occur at the following points:

- While waiting on a condition variable. That is, within `pthread_cond_wait()` or `pthread_cond_timedwait()`.
- While awaiting the end of another thread (`pthread_join()`).
- While testing specifically for a cancel request with `pthread_testcancel()`.
- While waiting for an asynchronous signal (`sigwait()`).
- When a thread is waiting within `pthread_delay_np()` (not a portable routine).
- During the timeslice interruption.
- When `pthread_setasynccancel()` is called, and either:
  - It has set the cancelability state to `asynchronous` (general cancelability and asynchronous cancelability are both enabled), it has not yet returned, and a cancel is pending.
  - It was called to disable `asynchronous` cancelability state, but has not yet done so, and a cancelation request has been asynchronously delivered.
- When suspended because of functions involving indefinite time periods, such as waits, sleep, or I/O. See [z/OS C/C++ Run-Time Library Reference](https://www.ibm.com/support/knowledgecenter/SSEPGG_11.3.0/com.ibm.zos.zos.doc/afs/afs5016r11.doc) under `pthread_setintrtype()` for more details.

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

If a thread has not set a `disabled` cancelability state, a cancelation request has been made to that thread, and the thread executes `pthread_testcancel()`, the cancelation request must be acted upon. Similarly, if a thread has not set a `disabled` cancelability state, a cancelation request has been made to that thread, and the thread is blocked at a cancelation point waiting for an event to occur, then that thread must act

<table>
<thead>
<tr>
<th>General Cancelability</th>
<th>Asynchronous Cancelability</th>
<th>Cancelability State</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANCEL_OFF</td>
<td>CANCEL_OFF</td>
<td>disabled</td>
</tr>
<tr>
<td>CANCEL_OFF</td>
<td>CANCEL_ON</td>
<td>disabled</td>
</tr>
<tr>
<td>CANCEL_ON</td>
<td>CANCEL_OFF</td>
<td>deferred</td>
</tr>
<tr>
<td>CANCEL_ON</td>
<td>CANCEL_ON</td>
<td>asynchronous</td>
</tr>
</tbody>
</table>
 upon the cancelation request. However, if a thread is suspended at a cancellation point and the event for
which it is waiting has completed before a cancelation request is received and acted upon, the thread may
resume normal execution and the cancelation request remains pending.

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

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

The side effects of acting upon a cancelation request while in a condition variable wait are:

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

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

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

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

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

```c
#include <pthread_exc.h>

<...>

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

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

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

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

    /* If the desired condition is not yet true, wait for */
    /* it to become true. This next call also auto- */
    /* matically releases the mutex... */
    pthread_cond_wait(some_object.condition, some_object.mutex);
```
Note that in order to handle the cancel as an exception, you must include the pthread_exc.h header file rather than pthread.h; this allows you to use the OSF DCE Threads exception interface.

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

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

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

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

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

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

- Applications should not use cancels as a synchronization mechanism. Condition variables should be used instead.
- pthread_mutex_lock() is not a cancelation point. Resources needing to be held exclusively for a long time should be protected by condition variables rather than mutexes, as this will not inhibit cancelability.
- A condition wait (via pthread_cond_wait() or pthread_cond_timedwait()) is a cancelation point. A side effect of acting on a cancelation request while in a condition wait is that the mutex is (in effect) reacquired. The effect is as if the thread were unblocked, allowed to execute up to the point of
returning from the wait, but at that point notices the cancelation request and handles it instead of returning.

- In general, most library calls cannot be assumed to be asynchronous cancel safe, and hence must not be called with cancelability state set to **asynchronous**.
- Cleanup routines should never exit via `longjmp()` or `siglongjmp()`.

In addition to the material covered in this section, "RPC Threads" on page 162 covers the additional semantics of cancels as applied to RPC threads.

### Signals

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

A signal is said to be **generated** for a process or thread when the event that causes the signal first occurs. Each process or thread has an action to be taken in response to each signal supported by the system or implementation. A signal is said to be **delivered** when the appropriate signal action for the process or thread is taken. A signal can be **blocked** (or **masked** by a thread or process by establishing a **signal mask** containing the signal to be blocked. The delivery of a blocked signal is deferred until it is unblocked. (Note that if the action specified for a signal is to ignore it, the signal effectively remains blocked.) During the time between its generation and delivery a signal is said to be **pending**.

Signals can be classified into two types with different semantics:

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

OSF DCE threads applications must handle synchronous and asynchronous signals differently.

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

Care must be taken when a thread unblocks an asynchronous signal. If another thread has blocked and is, or will be, waiting for the same signal, the results can be unpredictable and may result in the other thread waiting forever. This problem can be avoided by having all handling of asynchronous signals occur in a single thread, as described in "Asynchronous Signal Handling" on page 160."
Synchronous Signal Handling: Threads should call `sigaction()` to establish per-thread handlers for synchronous signals. The OSF DCE threads `sigaction()` function only modifies the signal action behavior for the calling thread and only works for synchronous signals. Threads must not use `sigaction()` for asynchronous signals.

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

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

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

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

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

    sigemptyset(&signals);
    /* In this sample, we'll catch only SIGINT... */
    sigaddset(&signals, SIGINT);
    sigprocmask(SIG_BLOCK, &signals, NULL);
    while(1)
    {
        sig = sigwait(&signals);
        switch(sig)
        {
            case SIGINT:
                /* SIGINT specific actions here. */
                break;
            default:
                /* Not reached. If we were waiting on other */
                /* signals, this would establish a default action */
                /* to exit... */
                continue;
                break;
        }
        sigprocmask(SIG_UNBLOCK, &signals, NULL);
        /* Do termination clean up here. */
    }
}
```

DCE Application Development Guide: Introduction and Style
exit(1);
}

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

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

Forking in a Threaded Application

Note: The following discussion does not apply to z/OS DCE threads, which does not support `fork()`.

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

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

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

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

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

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

**RPC Threads**

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

**RPC Cancel Semantics**

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

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

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

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

   This behavior contrasts with the local procedure call case: if cancel state is *disabled* when a local procedure call is made, and the callee sets the cancelation state to *deferred*, then if a cancel arrives and the callee hits a cancelation point, the cancel will be acted upon. Furthermore, if the cancel arrives after the callee returns, the cancel will be acted upon when a cancelation point is arrived at in the caller.

2. If cancelability state is *deferred*, then cancelation requests will be sent to the server where they will be handled according to the server’s setting of the cancelability state for the application thread extension (that is, the call thread) in the server. If ignored at the server, the client side would then effect the cancel upon return from the RPC, so the cancel would not be lost or incorrectly handled. In particular, the timeslice interrupt (context switch) is a cancelation point in OSF DCE threads, so that even if a cancel were ignored by the server side, when the RPC returns, the thread will be at a cancelation point.

3. If cancelability state is *asynchronous*, then cancelation can happen at any time. In general, this state is not recommended across the scope of an RPC in line with the rule that most routines that do useful work are not asynchronous cancel safe and thus should not be called with asynchronous cancelability state.
Chapter 5. Security

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

The roles of these three elements can be broadly defined as follows:

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

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

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

### The Basic Security Model

At a high level the DCE security model is as follows:

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

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

<table>
<thead>
<tr>
<th>Client Authentication Actions</th>
<th>Server Authentication Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rpc_c_authn_none</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>rpc_c_authn_dce_secret</strong></td>
<td>DCE secret key</td>
</tr>
<tr>
<td><strong>rpc_c_authn_default</strong></td>
<td>DCE secret key</td>
</tr>
</tbody>
</table>

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

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

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

### The Basic Security Model

At a high level the DCE security model is as follows:

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

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

<table>
<thead>
<tr>
<th>Client Authentication Actions</th>
<th>Server Authentication Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rpc_c_authn_none</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>rpc_c_authn_dce_secret</strong></td>
<td>DCE secret key</td>
</tr>
<tr>
<td><strong>rpc_c_authn_default</strong></td>
<td>DCE secret key</td>
</tr>
</tbody>
</table>
Authentication establishes only that each of the parties is a principal known to the authentication service, and that each party knows who the other is. Servers typically make an explicit authorization decision, using one of the DCE authorization services, to decide whether a given authenticated principal should in fact be granted access to some operation or resource. In most cases clients will not be satisfied with the mere assurance that they are communicating with an authenticated principal. Clients must then check the authenticated identity of the server to be sure that it is one with which they are willing to communicate. Note that this kind of server identity check is normally made at a low level of granularity; typically once per client-server session. Server authorization of clients is usually much more specific: typically once per remote operation.

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

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

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

In addition to authentication and authorization, the DCE security services can also provide various levels of data secrecy and integrity guarantees. The basic model is that the client application requests the minimum acceptable protection level. The runtime then provides the lowest supported protection level that is at least as high as the one requested by the client. If the runtime cannot provide at least the requested level, it fails the call. Supported levels, and the services provided by each level, depend on the authentication service in use, so clients must take care to request a level that is meaningful for the authentication service they have specified.

<table>
<thead>
<tr>
<th>Client Specifies</th>
<th>Server Registers</th>
<th>Authentication</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpc_c_authn_none</td>
<td>rpc_c_authn_none</td>
<td>No authentication performed</td>
</tr>
<tr>
<td>rpc_c_authn_none</td>
<td>rpc_c_authn_dce_secret</td>
<td>No authentication performed</td>
</tr>
<tr>
<td>rpc_c_authn_dce_secret</td>
<td>rpc_c_authn_none</td>
<td>Call rejected by RPC runtime</td>
</tr>
<tr>
<td>rpc_c_authn_dce_secret</td>
<td>rpc_c_authn_dce_secret</td>
<td>Authentication performed</td>
</tr>
</tbody>
</table>

Table 6. Authentication

DCE Application Development Guide: Introduction and Style
Application Roles

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

In the case of authentication, clients and servers have to do more work to establish the required state for authentication to take place. The required steps are described in detail in "Authorization" on page 176. Once this initialization is taken care of, the RPC runtime provides authentication transparently.

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

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

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

Authentication Model

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

The DCE Authentication Model

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

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

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

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

3. Authentication related RPC protocol activity is then carried out transparently by the RPC runtime during each call.

In addition, server application code needs to make authorization decisions based on the assumption that authentication has been carried out, but these belong more properly to the realm of authorization, as described in "Authorization" on page 176.

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

Application-Level Authentication

One of the obvious conclusions to be drawn from the general discussion of DCE authentication is that application-level client and server authentication responsibilities are highly asymmetrical: clients typically inherit identities while servers assume them implicitly; clients are concerned with credentials while servers are concerned with keys. The reasons for these asymmetries have to do both with the underlying asymmetry of the Kerberos model and with an underlying model of RPC client and server behavior that is also asymmetrical.

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

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

Clients, on the other hand, will typically be invoked by an interactive principal to run within the scope of a single principal login session. Such clients can therefore usefully acquire their credentials from the principal who invoked them. Note, however, that there is nothing to require clients to behave in this manner. A persistent client can easily be written that assumes its own identity, manages keys, and acquires and updates credentials. The basic authentication policies described here can be easily extended to cover this case.
For a client that runs with an inherited identity, the principal security problem— the maintenance of its secrets— is reduced to the problem of maintaining the security of its credentials while they are valid. The client is basically passive in this respect, depending on the local operating system to prevent unauthorized access to the credential cache of the DCE principal that initiates the client application. Direct management and discovery of keys (for example, reading them from a configuration file) is not required of such clients. Typically, such an application can do nothing about the security of the principal's keys used to acquire credentials, since all the authentication-related state is inherited. The client's real security responsibility is therefore negative: not to take any action outside of the specified authentication policy model that could compromise security for the identity with which it runs (for example, indiscriminately giving other processes access to its credentials).

Clients may or may not be concerned with the identities of the servers they call. The Kerberos authentication exchange is mutual in the sense that both clients and servers must have genuine authentication identities to participate successfully. However, a client may not trust a server simply because it can successfully authenticate to the client. The client may want to make RPCs only to servers with specific principal identities that it trusts. In this case, the client has the additional security task of safely maintaining a list of acceptable server identities with which it is willing to communicate.¹

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

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

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

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

Application steps for checking authenticated server identity are discussed in [Authorization on page 176]
Obtaining an Authentication Identity

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

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

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

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

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

```c
sec_login_setup_identity(prin_name, sec_login_no_flags, &login_context, status);
sec_key_mgmt_get_key(rpc_c_authn_dce_secret, keytab, prin_name, 0, (void**)&keydata,status);
sec_login_validate_identity(login_context, keydata, &reset_pwd, &auth_src, status);
```

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

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

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

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

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

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

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

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

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

Managing Keys

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

**Note:** Keytab files are normally created by administrative action. Be aware that the local identity or the process running `rgy_edit` determines the initial local ownership of files created by `ktadd`. When keytab files are created by `dcecp`, local ownership of the files is given to root.

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

Because the degree of integration between local and DCE login varies with DCE implementations, it is difficult to give more general advice about local identities. As the following paragraphs explain, however, it is generally not a good idea for the server to run with the DCE identity of a human user. If DCE and local identities are the same, then the same guideline must be applied to local identities. That is, the server's local identity should not be that of a human user.

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

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

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

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

**Default Server Authentication Steps**

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

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

The following sample functions, reproduced from the sample DCE application, implement credential acquisition, credential revalidation, and key management.

In order to save space and to improve the readability of the text, the code shown below has been slightly edited: all status checks, and all calls to the DCE serviceability interface (to print or log status or informational messages), have been removed.
**managekey:** The managekey() routine manages the server principal's key, making sure that it never expires.

```c
void managekey(char *prin_name){ /* Server principal name */
    unsigned32 status;
    status = error_status_ok;
    sec_key_mgmt_manage_key(
        rpc_c_authn_dce_secret, /* Authentication protocol. */
        KEYTAB, /* Local key file. */
        (idl_char *)prin_name, /* Principal name. */
        &status);
}
```

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

```c
void server_get_identity(char *prin_name)
```

```c
unsigned_char_p_t prin_name, /* Server principal name. */
sec_login_handle_t login_context, /* Returns server's login context. */
unsigned_char_p_t keytab, /* Local key file. */
unsigned32 *status)
{ }"
/* Create a context and get the login context... */
sec_login_setup_identity(prin_name,
    sec_login_no_flags,
    login_context,
    status);

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

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

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

}

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

/*****
* server_renew_identity -- Make sure that credentials are still valid, and
* renew them if they are not.
*
* This routine is called (with the current credentials) whenever a task
* is about to be attempted that requires valid credentials. For an ex-
* ample, see the cleanup code in "main()" above. A valid credential will
* nevertheless be considered invalid if it will expire within time_left
* seconds. This gives a margin of time between the validity check that
* occurs here and the actual use of the credential.
* Called from main() (but can be called from elsewhere).
* *******/

void server_renew_identity(
unsigned_char_p_t prin_name,  /* Server's principal name. */
sec_login_handle_t login_context,  /* Server's login context. */
unsigned_char_p_t keytab,  /* Local key file. */
unsigned32 time_left,  /* Amount of "margin" -- see above. */
unsigned32 *status)  /* To return status. */
{
    signed32 expiration;
    time_t current_time;
    sec_passwd_rec_t *keydata;
    sec_login_auth_src_t auth_src;
    boolean32 reset_pwd;

    *status = error_status_ok;
The server initialization code need then only make the following calls to establish server authentication and obtain valid credentials:

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

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

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

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

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

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

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

The following code fragment demonstrates this scheme.

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

boolean32 is_valid_principal(
    unsigned_char_t princ_name, /* Full name of principal to test. */
    unsigned_char_t group, /* Group we want principal to be in. */
    unsigned32 status) {         /* Local cell name. */
    sec_rgy_handle_t rhandle; /* Local registry binding. */
    boolean32 is_valid;        /* To hold result of registry call. */

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

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

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

    /* Free the cellname string space... */
    free(cell_name);
}
```
/* Get the specified principal's local (cell-relative) name... */
local_name = malloc(strlen((char *)princ_name));

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

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

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

}<....>

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

/* Find out what the server's principal name is... */
rpc_mgmt_inq_server_princ_name(binding_h,
  rpc_c_authn_dce_secret,
  &server_princ_name,
  &status);

/* And now find out if it's a valid member of our sample_servers */
/* group... */
if (is_valid_principal(server_princ_name, (unsigned_char_t *)SGROUP, &status))
{
  rpc_binding_set_auth_info(binding_h,
    server_princ_name,
    rpc_c_protect_level_pkt_integ,
    rpc_c_authn_dce_secret,
    NULL,
    rpc_c_authz_dce,
    &status);
}
Authorization

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

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

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

   If authentication information is provided, then the application should go on to ask:

2. Is the authentication service acceptable to the server? The application checks the `authn_svc` parameter. Currently this check is redundant, since the only authentication service available is DCE secret key (the `authn_svc` returned is `rpc_c_authn_dce_secret`).

   The server may of course, simply be satisfied that the client is authenticated and check no further. Or the server can:

3. Check that the protection level is acceptable. This too is a matter for negotiation between the client and server applications, but it is important to begin by considering the runtime's mediation of the protection level request. Recall that the client specifies a specific protection level for a binding, whereas the server, when it registers its authentication information, specifies only the authentication service it will use.

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

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

4. Check that the authorization service is acceptable. Once again, this is a matter for negotiation between the client and server applications. The server application provides an access testing mechanism for authorization services it supports. There are three possibilities:

   - Authorization based on the client's principal name (`rpc_c_authz_name`)

There is considerable asymmetry in the use of the `authn_svc` values on the client call to `rpc_binding_set_auth_info()` and the server call to `rpc_binding_inq_auth_client()`. If the client specifies `rpc_c_authn_none`, then the server sees a status of `rpc_s_binding_has_no_auth`, and no meaningful value is returned for the `authn_svc` parameter. Furthermore, given that the default authentication service is DCE secret key, if the client specifies `rpc_c_authn_default`, the server returns `rpc_c_authn_dce_secret` from `authn_svc`. In other words, while the client can specify three different values for `authn_svc`, the server can return only one.
Authorization based on the client's credentials (rpc_c_authz_dce). This involves checking the client identity's permission set (extracted from an Access Control List (ACL) associated with the object the client is attempting to access) against the required permissions for the requested operation. The client's identity is extracted from its credentials, contained in its binding.

The server may permit access without authorization checking (rpc_c_authz_none).

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

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

If the authorization service requested is acceptable, the server application makes the appropriate access tests as described in step 6.

5. Check that the server principal name specified by the client is acceptable. This check is useful for a server that is running with more than one principal identity. The server may only want to allow the operation under a specific principal identity. If the server is running with only one principal identity, then this check is redundant.

6. Extract the client privileges and perform the appropriate access testing. The form of the client privileges depends on the authorization service. The application needs to extract the privileges in the correct format and pass them to the appropriate access tests.

Authorization using RACF-DCE Interoperability

When developing or modifying new or existing applications, you may want to use DCE security for client authentication and a communications paradigm, such as DCE RPC (authenticated) or APPC (using GSSAPI) for segmenting the client and server portions of the application.

An application developer may wish to use the authorization and auditing capabilities provided by RACF for the server portion of a client-server application that resides on z/OS. In this case, the application requires a mechanism to associate the occurrence of a DCE principal with the corresponding z/OS user ID.

RACF provides support for z/OS DCE through enhancements to the RACF user profile, and the addition of a new general resource class. These enhancements enable RACF to maintain a subset of the DCE principal information that is contained in the DCE registry. Through extensions to the RACF user profile, and with the addition of a general resource class, RACF can maintain a linkage between a DCE principal, and a z/OS user ID. The linkage between a DCE principal and a RACF user ID is known as cross-linking. The cross-linking information placed in a RACF user profile DCE segment and the RACF general resource class, DCEUUIDS, provides this information.
Using Cross Linking Information:  How does an application make use of DCE security and the cross-linking information in RACF to obtain the z/OS user ID? The server does these steps:

1. The client and server must be at the DCE 1.1 level and be using DCE security.

2. The server application issues the `rpc_binding_inq_auth_caller()` and gets an opaque handle returned that represents the client's authorization information.

3. The server application then issues `sec_cred_get_initiator()`, passing the opaque handle obtained from the `rpc_binding_inq_auth_caller()`. Another opaque handle to a single DCE principal's identity authorization information is returned. If DCE delegation has been used, there may be other identities in the handle. (See [z/OS DCE Application Development Guide: Core Components](https://www-01.ibm.com/support/docview.wss?uid=swg27047311) for a discussion on delegation.) Therefore, this may or may not be the handle for the authentication information for the client that issued the call to the application server on z/OS. If it is not the correct handle, and if the identity information exists for the intermediate delegates of the client, another call must be made to retrieve a handle to the next entry. The process for parsing the handle is discussed in step 5.

4. The z/OS application server then calls `sec_cred_get_pa_data()` passing in the opaque handle obtained from `sec_cred_get_initiator` to get the Privilege Attribute Certificate (PAC) or Extended Privilege Attribute Certificate (EPAC) for the initiator. This contains the principal name, principal UUID and cell UUID.

5. The above assumes no delegation is involved. If the client is using delegation:
   - a. The server calls `sec_cred_get_delegate` in a loop until it returns `sec_cred_no_more_entries`.
   - b. If the call returns an opaque handle, `sec_cred_get_pa_data()` is called passing the opaque handle in as input to retrieve the PAC or EPAC. The identity of each delegate in the chain is checked this way.

6. After obtaining a principal's UUID and cell UUID, which are in binary format, the application server invokes the `uuid_to_string()` function, passing the principal's UUID on input. This returns the string format of the UUID. Invoke this API again to convert the cell UUID. For more information on the above DCE APIs, see [z/OS DCE Application Development Reference](https://www.ibm.com/support/docview.wss?uid=swg27047311).

Using z/OS DCE and RACF Services  Once the DCE client's UUID is known, the z/OS application server has a number of choices of how to use z/OS DCE and RACF services for resource authorization:

- RACF maintains a subset of the DCE principal information that can be used to convert a DCE identity (defined by DCE UUID) to a z/OS user ID. The z/OS application server can convert the DCE UUID to a z/OS user ID by invoking the `R_dceruid` SAF callable service, z/OS UNIX `convert_id_up()`, or `__convert_id_np()` C library function call.

- If the server is Authorized Program Facility (APF) authorized, or capable of executing in supervisor state or system key (0-7), the z/OS application server can use the SAF services RACROUTE macro instruction do any of the following:
  - Create and delete a RACF security context
  - Use SAF services to check accesses for resources that the z/OS application server manages.
  - Customize a task (task control block) with the RACF identity of the client so decisions on resource access are based on the identity of the client.

- If the server application is not APF authorized and does not execute in supervisor state or a system key, the z/OS application server can do any of the following:
  - Obtain the z/OS user ID associated with the client UUID by using the z/OS UNIX `convert_id_up()`, or the C library function call `__convert_id_np()`.
  - Customize a thread with the identity of the DCE client, using the `pthread_security_np()` C library call
– Use the __check_resource_auth_np() C library function call (if the z/OS application server owns the resource to which the client is requesting access) to centralize access control decisions in RACF.

See z/OS DCE Application Development Guide: Core Components and z/OS DCE Application Development Reference and IBM C/C++ publications for more information on these APIs.

Other Uses for the Cross Linking Information: z/OS DCE provides support, called single sign-on, which allows a user to log on to z/OS and invoke DCE applications without reauthenticating to DCE. The cross-linking information in the RACF DCE segment is used to log the DCE principal in to DCE. For more information on single sign-on, see z/OS DCE Administration Guide and z/OS DCE User's Guide.

Client Credentials

A client's credentials may be implicitly passed on to an ACL manager via a call to dce_acl_is_client_authorized(). Or the credentials may be extracted from the client binding by a call to rpc_binding_inq_auth_client() and then passed on to an ACL manager via a call to sec_acl_mgr_is_authorized(). In the latter case, there is some additional complication in the case that the client specified no authentication. If the server supports credential-based authorization, it should handle this case by testing for unauthenticated access via the ACL manager. However, no credentials are returned from rpc_binding_inq_auth_client() in this case. The convention is to set the pac argument to NULL in this case ((rpcauthzhandle_t)0). ACL managers that follow the recommended policies will test for unauthenticated access in the case of such a null handle.

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

The following code fragment shows the necessary steps:

```c
rpc_authz_handle_t pac;

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

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

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

Authorization decisions depend on the following information:

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

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

**ACL privilege attribute entries:**

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

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

**ACL mask entries:**

These consist of two *entry type:permissions set* pairs.

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

There are two levels of semantics/policy to be considered here. One is the semantics of privilege attributes, for which we specify a strict (POSIX compliant) policy in the form of an access checking algorithm. This is embodied in the default access checking algorithm provided by the ACL library. The second is the semantics of permissions. Ultimately these depend on the ACL manager and the kinds of objects it protects. However, some recommendations for keeping permissions as intuitive and consistent across applications as possible are offered in the following subsection.

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

1. *user_obj*
2. *user*
3. *foreign_user*
4. *(group_obj), group, foreign_group*
5. *other_obj*
6. *foreign_other*
7. *any_other*

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

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

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

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

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

The user_obj and group_obj types have less straightforward semantics. They refer to a special principal and group that must be known to the ACL manager “out of band”: that is, they cannot be determined from the ACL entry itself. The semantics of the mask_obj, which is applied to everything except the user_obj and other_obj entries, are also complicated. The mask_obj is implemented to permit POSIX ACLs to more or less maintain UNIX semantics for 000 permissions.

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

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

1. user
2. group
3. other_object
4. foreign_user
5. foreign_group
6. foreign_other
7. any_other

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

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

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

The DCE ACL library is an implementation of the remote ACL (rdacl) interface, designed in such a way as to allow any DCE application to use it instead of having to implement the interface itself. In DCE 1.0, applications that wished to use the DCE ACL functionality had to implement the full remote interface themselves; in DCE 1.1 this is no longer true. Once an application has registered certain information with the ACL library (see step 2 in "The Requirements" on page 183), its ACL management information will be hooked into the remote ACL implementation routines that the DCE ACL library consists of.

Of course, an application still must take care of the details of storing and retrieving its ACLs (though these tasks are now made much easier by the DCE Backing Store library routines), setting up definitions that determine how its ACLs are interpreted, and so on. Practical examples of how to do these things can be found in the DCE example application sample, which is explained in the following sections.

For more detailed information about the interfaces mentioned below, see the information about the DCE Backing Store and the Access Control List Application Program Interface in the z/OS DCE Application Development Guide: Core Components.

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

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

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

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

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

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

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

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

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

1. There must be a procedure that can take the valid name of an object and return that object's ACL 
   UUID to a caller. This typically is accomplished by (first) looking up an object UUID in a 
   name-indexed database and then (secondly) extracting the ACL UUID from the object state 
   information, which was looked up in a database indexed by object UUIDs. The databases, of course, 
must be set up and maintained by the application. Clients to bind to the objects through a CDS  
"junction" at the server's entry.

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

   This is the server object name resolution procedure described in item 1. `acl_edit` accepts a (CDS 
entry) name which it expects to be able to resolve into an object which has an ACL it can access. For 
this to be so, the application server must register a routine (with the rdacl interface UUID) which, 
when called by `acl_edit` with an object name, will be able to return to `acl_edit` the information it needs 
(that is, a UUID) to get the ACL itself. In other words, the routine must be able to turn an object name 
into an ACL UUID.

3. A persistent database in which to store the ACLs must be created. (The database must be compatible 
with the interface that the Security routines use; that is, it must be created with the DCE Backing Store 
library routines.)

4. The ACL database must be registered (together with a manager type UUID and a name-to-ACL UUID 
resolver procedure) with the ACL library.

5. An "object type" (that is, manager type) UUID must be created to identify each of the application's ACL 
categories (that is, the kind of object the ACL applies to, and hence the kind of ACL itself: what 
permissions it can contain, and what they mean in regard to the object they protect). (The manager 
type will also serve as an identifier for the ACL database that the ACLs themselves are stored in — 
this however is internal to the ACL library.)

6. UUIDs to identify the objects must be created.

7. The ACLs themselves on the relevant objects must be created.

8. The ACLs must be stored, indexed by UUID, in the Backing Store database.

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

Chapter 5. Security 183
Note, by the way, that discussing the details of setting up an ACL manager without first considering the representation and management of the objects themselves is a very artificial thing to do. The excuse for doing it here is that ACL managers are the subject of this section. However, keep in mind that ACLs are only an adjunct to the objects they guard access to. In a real application one would never put the cart before the horse by working out the details of ACL management before settling on the way object management itself was to be done.

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

In any application, distributed or not, an “object” is any externally accessible resource which is under the application's control. Objects can be anything: printers, files, other machines, data — it all depends on the application. What these things have in common is that they must be accessed through the application itself. Entities in a distributed application request the use of these resources, via clients, from the application server; and the server normally decides whether or not to grant use of a resource to an entity by examining the object's ACLs.

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

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

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

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

Secondly: There is a third database to hold only the objects' ACLs. Theoretically speaking, there is no reason why the ACLs couldn't be held with the rest of the objects' information, in the object UUID-indexed database. However, the application's ACLs must be accessible to the DCE ACL library routines, and these routines expect a database, indexed by ACL UUIDs, containing only ACLs. This allows us, for example, to call a DCE routine such as **dce_acl_is_client_authorized()** (see the **sample_mgmt_auth()** callback routine in **sample_server.c**), passing the ACL manager type UUID and the ACL UUID, and get...
back an answer to some query about permissions — the library routine is able to go into the database and access and read the ACL; we don't have to bother with that. It also allows the rdacl implementations in the ACL library to do the same thing, since they have a full ACL binding (which includes a handle to the database in which the ACL is stored).

**Object Name Resolution Routine:** Our application’s “name-to-ACL UUID” resolution routine uses the following algorithm:

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

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

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

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

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

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

A practical sense of what this means can be had from looking at the two managers’ ACL printstrings, near the top of the sample_server.c file. These strings, which contain text representations of the full range of permissions supported by the respective managers, show that there are many permissions that are unique to a single manager. For example, there is a m_inq_if permission (permission to execute the rpc_mgmt_inq_if_ids() routine against the server). This permission makes sense only in the context of the server_mgmt object. A manager type identifies what set of permissions applies to a given set of objects.
How the ACL Library Routines Extract and Evaluate ACLs: One way of using ACLs to evaluate an entity's authorization to do something is by making a call to the DCE library routine `dce_acl_is_client_authorized()`. For example, there are two places in the example application where this is done to check client access to the application's own objects:

- in `sample_call()` (in `sample_manager.c`)
  This is an interface operation, called by the client.
- in `sample_mgmt_auth()` (in `sample_server.c`)
  This is the remote management callback function.

(Similar routines are called remotely through the `sec_acl...()` routines.)

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

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

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

How does the library routine (especially when it is called, not from "inside" the application as noted at the beginning of this section, but, say, by `acl_edit`) know how to access the correct ACL database from which to extract and examine the ACL identified by the ACL UUID? The answer is that the application's database will have become known to the caller in the course of establishing a binding to the server. (This is done by calling the application's registered "resolver" routine; the library finds the right resolver routine by calling all the attempted resolvers that have been registered with it until it gets a successful return. It finds the ACL manager type in the same way, since it calls each attempted resolver passing the manager type UUID that was registered with it. See the `sample_resolve_by_name()` function in the `sample_server.c` file.)

Backing Store Database Items and Headers: (Note that although backing stores are necessary in implementing an ACL manager, their use is not limited to ACL management. Backing stores are designed to be used for all kinds of persistent storage of distributed data. For more information about the DCE Backing Store, see the [z/OS DCE Application Development Guide: Core Components](https://www.ibm.com/support/knowledgecenter/SSDTEP_1.7.0/com.ibm.zos.dce.nk01.doc/bas下发.del) )

As mentioned earlier, backing store databases are necessary for storing any information about the application's objects that must be preserved between application server sessions. The example application uses three such databases, as described in "Object Name Resolution Routine" on page 185.

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

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

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

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

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

- Object itself
- Owner of the object
- Owner’s group
- Object’s ACL
- Default object ACL
- Default container ACL

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

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

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

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

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

Note that this is the database that the ACL library has to have access to; this access is set up by a call to `dce_acl_register_object_type()`, which registers a manager type + database + resolver_routine combination. The registration then allows the ACL library to derive any or all of these three things from an object name (the application’s resolver routine has to help out in this, of course).
ACL Manager Coding Example: The following subsections contain extracts from the DCE sample application sample. The subsections below contain only the ACL manager code portions of the server application.

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

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

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

//#define mgmt_perm_inq_if sec_acl_perm_unused_gz'ro@otgz'ro@otgz'ro@otgz'ro@otgz'ro@ot8gz'ro@ot
#define mgmt_perm_inq_pname sec_acl_perm_unused_gz'ro@otgz'ro@otgz'ro@otgz'ro@otgz'ro@ot1gz'ro@ot
#define mgmt_perm_inq_stats sec_acl_perm_unused_gz'ro@otgz'ro@otgz'ro@otgz'ro@otgz'ro@ot2gz'ro@ot
#define mgmt_perm_ping sec_acl_perm_unused_gz'ro@otgz'ro@otgz'ro@otgz'ro@otgz'ro@ot4gz'ro@ot
#define mgmt_perm_kill sec_acl_perm_unused_gz'ro@otgz'ro@otgz'ro@otgz'ro@otgz'ro@ot8gz'ro@ot

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

uuid_t mgmt_acl_mgr_uuid = { 0x0060f928-bbf3-1d35-8d7d-0000c0d4de56,
                              0x0060f928, 0xbbf3, 0x1d35, 0x8d7, 0x000, 0x000, 0x0d4, 0x0de, 0x56 };

uuid_t mgmt_object_uuid = { 0x00573bc0e-bcc2-1d35-a73e-0000c0d4de56,
                              0x00573bc0e, 0xbcc2, 0x1d35, 0xa7, 0xe3, 0x00, 0x000, 0x0x0c, 0x0d4, 0x0de, 0x56 };

// These UUIDs are specific to this server...

uuid_t mgmt_acl_uuid;

uuid_t sample_acl_uuid;

// The UUID of the sample ACL manager:

uuid_t sample_acl_mgr_uuid = { 0x01a15a9-3382-1d23-a16a-0000c0d4de56,
                              0x001a15a9, 0x3382, 0x1d23, 0xa1, 0x6a, 0x00, 0x000, 0x0c0, 0x0d4, 0x0de, 0x56 };

// A UUID for a sample object:

uuid_t sample_object_uuid = { 0x0415371-f29a-1d3d-b8c8-0000c0d4de56,
                              0x00415371, 0xf29a, 0x1d3d, 0xb8c8, 0xc8, 0x00, 0x000, 0x0x0c, 0x0d4, 0x0de, 0x56 };

// The mgmt printstrings could be treated as standard for a standard mgmt ACL manager...

sec_acl_printstring_t mgmt_printstr[] = {
    { "i", "m_inq_if", mgmt_perm_inq_if },
    { "n", "m_inq_pname", mgmt_perm_inq_pname },
    { "s", "m_inq_stats", mgmt_perm_inq_stats },
    { "p", "m_inq_perm", mgmt_perm_ping },
    { "k", "m_perm_kill", mgmt_perm_kill },
    { "t", "m_perm_test", mgmt_perm_test },
    { "c", "m_perm_control", mgmt_perm_control }
};
```

188  DCE Application Development Guide: Introduction and Style
});

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

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

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

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

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

void server_get_local_principal_id(
    unsigned_char_t *p_name,  /* Simple principal name. */
    uuid_t *p_id,             /* UUID returned here. */
    unsigned32 *status)       /* Status returned here. */
{
    char *cell_name;          /* For local cell name. */
    sec_rgy_handle_t rhandle; /* For registry server handle. */

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

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

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

    /* Now get from the registry the UUID associated with the principal */
    /* name we got in the first place... */
    sec_rgy_pgo_name_to_id(rhandle,
        sec_rgy_domain_person,
        p_name,
        p_id,
        status);
}
server_create_acl: The server_create_acl() routine creates an ACL for a specified principal.

(/*****
 * server_create_acl -- Create an ACL with some specified set of permissions
 * assigned to some principal user.
 *
 * Called from server_acl_mgr_setup().
 *
******/

void server_create_acl(
    uuid_t mgr_type_uuid, /* Manager type of ACL to create. */
    sec_acl_permset_t perms, /* Permission set for ACL. */
    unsigned_char_t *user, /* Principal name for new entry. */
    sec_acl_t *acl, /* To return the ACL entry in. */
    uuid_t *acl_uuid, /* To return the ACL's UUID in. */
    unsigned32 *status) /* To return status in. */
{
    uuid_t u; /* For the principal's UUID (from the registry). */

    *status = error_status_ok;

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

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

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

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

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

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

void server_store_acl(
    dce_db_handle_t db_acl, /* ACL (UUID)-indexed store. */
    dce_db_handle_t db_object, /* Object (UUID)-indexed store. */
    dce_db_handle_t db_name, /* Name-indexed store. */
    sec_acl_t *acl, /* The ACL itself. */
    uuid_t *acl_uuid, /* ACL UUID. */
    unsigned32 *status) /* To return status in. */
{...}
uuid_t *object_uuid, /* Object UUID. */
unsigned_char_t *object_name, /* The name of the object. */
void *object_data, /* The actual object data contents. */
/* NOTE: NOT USED NOW. */
boolean32 is_container, /* Are we storing a container ACL? */
unsigned32 *status) /* To return status. */
{
/* These two variables are used to hold UUIDs for the ACLs we will */
/* need to create if we have a container ACL on our hands... */
uuid_t def_object, def_container;
sample_data_t sample_data;

*status = error_status_ok;

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

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

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

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

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

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

/* Store the ACL UUID(s) into a standard object header... */
dce_db_std_header_init(
    db_object, /* Object database. */
    &(sample_data.s_hdr), /* Object data hdr. */
    object_uuid, /* Object UUID. */
    acl_uuid, /* ACL UUID. */
    &def_object, /* Default object ACL. */
    &def_container, /* Default container ACL. */
    0, /* Reference count. */
    status);

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

dce_db_store_by_uuid(db_object, object_uuid, (void *)&sample_data, status);
server_acl_mgr_setup: The server_acl_mgr_setup() routine performs all the steps necessary to set up ACL databases for the two object types used by the sample application.

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

void server_acl_mgr_setup(
    unsigned_char_t db_acl_path, /* Pathname for databases. */
    dce_acl_resolve_func_t resolver, /* sample_resolve_by_name. */
    uuid_t acl_mgr_uuid, /* ACL manager UUID. */
    uuid_t object_uuid, /* Object UUID. */
    unsigned_char_t object_name, /* Object name. */
    sec_acl_permset_t owner_perms, /* Owner permission set. */
    unsigned_char_t owner, /* Owner name. */
    boolean32 is_container, /* Is this a container object? */
    /* == TRUE from main(). */
) {
    sec_acl_t new_acl;
    uuid_t machine_princ_id;
    unsigned_char_t машине[20];
    unsigned_char_t *uuid_string;
    boolean32 need_init;
    unsigned32 dbflags;
    static sample_data_t datahdr;
    unsigned_char_t acl_path_string;
    sec_acl_permset_t *permset = (sec_acl_permset_t) 0;
    *status = error_status_ok;
    bzero(&datahdr, sizeof datahdr);
    uuid_create_nil(object_acl_uuid, status);
    need_init = 0;
    /* Build the full pathname string for the db_acl database... */
}
acl_path_string = malloc(MAX_ACL_PATH_SIZE);
strcpy(acl_path_string, db_acl_path);
strcat(acl_path_string, (unsigned_char_t *)"/");
strncat(acl_path_string, "db_acl", strlen("db_acl"));

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

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

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

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

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

/* For the object database, we need standard backing store headers */
if (need_init)
    dbflags |= db_c_std_header;

/* Now open (or create) the "db_object" store... */
/* Build the full pathname string for the database... */
free(acl_path_string);
acl_path_string = malloc(MAX_ACL_PATH_SIZE);
strcpy(acl_path_string, db_acl_path);
strcat(acl_path_string, (unsigned_char_t *)"/");
strncat(acl_path_string, "db_object", strlen("db_object"));

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

/* Create the indexed-by-name database... */
if (need_init)
    dbflags |= db_c_create;
/* Build the full pathname string for the database... */
free(acl_path_string);
acl_path_string = malloc(MAX_ACL_PATH_SIZE);
strcpy(acl_path_string, db_acl_path);
strcat(acl_path_string, (unsigned_char_t)"/");
strncat(acl_path_string, "db_name", strlen("db_name"));

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

free(acl_path_string);

******************************************************************************

/* Now register our ACL manager's object types with the ACL */
/* library... */
/* Register for the mgmt ACL... */
dce_acl_register_object_type(
    *db_acl, /* Backing store where ACLs are to be stored. */
    &mgmt_acl_mgr_uuid, /* Type of ACL manager: this one is */
    /* for mgmt ACL operations; the UUID is defined */
    /* globally at the top of this file. */
    /* Why do we need this parameter? Well, the way */
    /* that the ACL library keeps track of the differ-
        * /
    /* ent "sets" of ACL databases is by manager UUID. */
    /* The manager UUID is what the library will use */
    /* to figure out which ACL database to open and */
    /* retrieve a requested ACL's contents from. */
    /* Essentially what we are doing here is setting */
    /* up things so that calls to the library routine */
    /* dce_acl_is_client_authorized() can be made to */
    /* check our ACLs, giving only the ACL UUID and a */
    /* manager UUID to get the desired result. */
    sizeof mgmt_printstr/sizeof mgmt_printstr[0], /* Number of */
    /* items in mgmt_printstr array. */
    mgmt_printstr, /* An array of sec_acl_printstring_t struc-
        * /
    /* tures containing the printable repre-
        */
    /* sentation of each specified permission. */
    &mgmt_info, /* A single sec_acl_printstring_t contain-
        */
    /* ing the name and short description for */
    /* the given ACL manager. */
    sec_acl_perm_control, /* Permission set needed to change */
    /* an ACL. Constants like these are defined */
    /* in <dce/aclbase.h>. */
    sec_acl_perm_test, /* Permission set needed to test an ACL. */
    resolver, /* Server function to get ACL UUID for a given */
    /* object; for us it's the */
    /* sample_resolve_by_name() call, below. */
    /* This routine is for the use of acl_edit: */
    /* it allows acl_edit to receive an object */
    /* name and come up with the ACL UUID; at */
    /* least that's what I think it's for. */
    NULL, /* Flags -- none here. */
    0,
);
/* Now register for the regular ACL... */
dce_acl_register_object_type(
    &db_acl, /* Backing store where ACLs are to be stored. */
    &sample_acl_mgr_uuid, /* Hard-coded at the top of this */
    /* file. */
    sizeof sample_printstr/sizeof sample_printstr[0], /* Number */
    /* of items in our printstring array. */
    sample_printstr, /* An array of sec_acl_printstring_t */
    /* structures containing the printable rep- */
    /* resentation of each specified permis- */
    /* sion set. */
    &sample_info, /* A single sec_acl_printstring_t contain- */
    /* ing the name and short description for */
    /* the manager we're registering. */
    sec_acl_perm_control, /* Permission set needed to change an */
    /* ACL. */
    sec_acl_perm_test, /* The permission you need to test an */
    /* ACL maintained by this manager. */
    resolver, /* Application server function that gives */
    /* the ACL UUID for a given object, when */
    /* presented with that object's name; for */
    /* us it's the sample_resolve_by_name() */
    /* routine, below. */
    NULL, /* Argument to pass to resolver routine; */
    /* identified as the "resolver_arg" in the */
    /* code to that function below. */
    0, /* Flags -- none here. */
    status);

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

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

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

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

    /* Get the machine principal's UUID... */
    server_get_local_principal_id(
        machine_principal,
        &machine_princ_id,
        status);

    /* Add a user entry for the machine principal to the new */
    /* ACL... */
    permsgset = ALL_MGMT_PERMS;
    dce_acl_obj_add_user_entry(
/* By default everybody must be able to get the principal name. They should be able to ping too. So add an appropriate unauthenticated permissions entry to the ACL... */
permset = mgmt_perm_inq_pname | mgmt_perm_ping;
dce_acl_obj_add_unauth_entry(
    &new_acl,
    permset,
    status);

/* Add permissions for the any_other entry in the ACL... */
permset = mgmt_perm_inq_pname | mgmt_perm_ping;
dce_acl_obj_add_any_other_entry(
    &new_acl,
    permset,
    status);

/* Store the mgmt ACL... */
server_store_acl(
    *db_acl, /* The ACL UUID-indexed store. */
    *db_object, /* The object UUID-indexed store. */
    *db_name, /* The name ("residual")-indexed store. */
    &new_acl, /* The ACL itself. */
    mgmt_acl_uuid, /* The mgmt ACL UUID. */
    &mgmt_object_uuid, /* The mgmt object UUID. */
    (unsigned_char_t *)MGMT_OBJ_NAME, /* The mgmt object name. */
    0, /* Not a container ACL. */
    status);

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

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

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

/* Store the object ACL... */
server_store_acl(
    *db_acl, /* The ACL UUID-indexed store. */
    *db_object, /* The object UUID-indexed store. */
    *db_name, /* The name ("residual")-indexed store. */
    &new_acl, /* The ACL itself. */
    object_acl_uuid, /* The object ACL UUID. */
    &object_uuid, /* The object UUID. */
    object_name, /* The object name. */
    0, /* Is this a container */
    &datahdr, /* The data header = object contents. */
    status);
/ * ACL? */

status);

/* Finally, free the space we were using... */
dce_acl_obj_free_entries(&new_acl, status);

/* ...end of object ACL creation code. */
*******************************************************************************/

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

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

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

} /*resolver)(

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

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

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

```c
void server_acl_mgr_close(
    dce_db_handle_t db_acl, /* ACL UUID-indexed database. */
    dce_db_handle_t db_object, /* Object UUID-indexed database. */
    dce_db_handle_t db_name, /* Name-indexed database. */
    unsigned32 status)
{
    *status = error_status_ok;
    dce_db_close(db_acl, status);
    dce_db_close(db_object, status);
    dce_db_close(db_name, status);
}
```

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

```c
void server_rdacl_export(
    rpc_binding_vector_t *binding_vector, /* Binding handles from RPC runtime. */
    uuid_vector_t *object_uuid_vector, /* Server instance UUID(s). */
    unsigned32 *status)
{
    uuid_vector_t my_uuids;
    *status = error_status_ok;
    /* Register the server's endpoints with the rdacl interface at the */
```
server_rdacl_cleanup: The server_rdacl_cleanup() routine removes the remote ACL interface information from the local endpoint map.

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

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

        rpc_ep_unregister(rdaclif_v1_0_s_ifspec,
            binding_vector,
            object_uuid_vector,
            status);
    }

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

    /********
    * sample_mgmt_auth -- Management authorization callback function.
    *
    * This is the routine that is implicitly called to test authorization
    * whenever someone tries to use the mgmt interface to tinker with us
    * or our ACLs.
    *
    * The callback is set up by a call to rpc_mgmt_set_authorization() in
    * server_acl_mgr_setup().
    *
    /*******

    boolean32 sample_mgmt_auth(
        rpc_binding_handle_t client_binding, /* Client's binding, whoever he is. */
        unsigned32 requested_mgmt_operation, /* What client is attempting to do. */
        unsigned32 *status)
    {
        boolean32 authorized = 0;
        sec_acl_permset_t perm_required;
        unsigned_char_t uuid_string;

        *status = error_status_ok;
    }
switch (requested_mgmt_operation) {
  case rpc_c_mgmt_inq_if_ids:
    perm_required = mgmt_perm_inq_if;
    break;
  case rpc_c_mgmt_inq_princ_name:
    perm_required = mgmt_perm_inq_pname;
    break;
  case rpc_c_mgmt_inq_stats:
    perm_required = mgmt_perm_inq_stats;
    break;
  case rpc_c_mgmt_is_server_listen:
    perm_required = mgmt_perm_ping;
    break;
  case rpc_c_mgmt_stop_server_listen:
    perm_required = mgmt_perm_kill;
    break;
  default:
    /* This should never happen, but just in case... */
    return(gzro@ot);
}

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

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

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

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

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

    *st = error_status_ok;

    /* Get the object's UUID, which will be the key that we will use to */
    /* fetch this particular object's data in the call following this */
    /* one... */
    dce_db_fetch_by_name(db_name, (char *)name, /* (void *) */ &u, st);
    up = &u; /* ...take the pointer to the key. */

    /* Using the UUID "key" that we just retrieved, get the dataheader */
    /* for the desired object (note that the data that one retrieves */
    /* with this routine can be anything; it depends on what we are */
    /* using the backing store for)... */
    dce_db_fetch_by_uuid(db_object, up, /* (void *) */ &dataheader, st);

Chapter 5. Security 201
/* Now, depending on the kind of ACL we're hunting for (i.e. ob-
object, container, etc.), extract its UUID from the object's
header structure... */
switch (sec_acl_type)
{
    case 1:
        acl_uuid = dataheader.s_hdr.tagged_union.h.def_object_acl;
        break;
    case 2:
        acl_uuid = dataheader.s_hdr.tagged_union.h.def_container_acl;
        break;
    default:
        acl_uuid = dataheader.s_hdr.tagged_union.h.acl_uuid;
}

/* Here it might be interesting to try retrieving the ACL itself, */
/* and e.g seeing what its manager type is... */
dce_db_fetch_by_uuid(db_acl,
    acl_uuid,
    &retrieved_acl,
    st);

/* We are handling two ACL managers through this function, so we */
/* have to make sure that we've extracted from the single ACL */
/* database the correct ACL: i.e., one whose manager type UUID is */
/* identical to the manager_type parameter we were passed: this is */
/* the manager whose ACL the runtime is trying to bind to. So... */
if ((manager_type != NULL) && (uuid_equal(manager_type, &(retrieved_acl.sec_acl_manager_type), st)))
{
    /* Return a bad status... */
    *st = acl_s_bad_manager_type;
    /* And no ACL UUID... */
    acl_uuid = NULL;
    return(0);
}

}
Chapter 6. Binding

Binding is the process by which an RPC client establishes a relationship with a server that supports an interface, object or some other resource the client is interested in. Since clients operate on server held resources by making RPCs, binding can be thought of, specifically, as creating the state required for an RPC to be made. In practice, the work of binding clients to servers normally involves name and endpoint mapping services. Strictly speaking, however, neither of these services is required for binding, since well-known bindings and endpoints can be used (in the form of string bindings). This chapter discusses the underlying binding model, apart from the use of name and endpoint services. It forms an essential introduction for the discussion of name and endpoint services that follows in Chapter 7, “Using the DCE Name Service” on page 215.

The Binding Model

Binding refers to the establishment of a relationship between a client and a server that permits the client to make a remote procedure call to the server. The term “binding” usually refers specifically to a protocol relationship between a client and either the server host or a specific endpoint on the server host, and “binding information” means the set of protocol and addressing information required to establish such a binding. But, for a remote procedure call, such a binding occurs in a context that involves other important elements, paralleling the notion of a binding in a local procedure call. In order for an RPC to occur, a relationship must be established that ties a specific procedure call on the client side with the manager code that it invokes on the server side. This requires both the binding information itself and a number of additional elements (see Figure 119 on page 204). The complete list is as follows:

1. A protocol sequence that identifies the RPC and underlying transport protocols
2. An RPC protocol version identifier
3. A transfer syntax identifier
4. A server host network address
5. An endpoint of a server instance on the host
6. An object UUID that can optionally be used for selection among servers and/or manager routines
7. An interface UUID that identifies the interface to which the called routine belongs
8. An interface version number that defines compatibility between interface versions
9. An operation number that identifies a specific operation within the interface
The binding information itself covers the first five elements of the list — the protocol and address information required for RPC communications to occur between a client and server. (Figure 119 also shows the object UUID as part of the binding information. This applies to clients, as explained in "Client Binding Model" on page 205.) In RPC terminology, such a binding can be partial or full. A partial binding is one that contains the first four elements of the list, but lacks an endpoint. A full binding contains an endpoint as well. The distinction is that a partial binding is sufficient to establish communications between a client and a server host, whereas a full binding allows communications to a specific endpoint on the server host.

In order to complete an RPC call, all of the elements listed in 119 except the object UUID must be present. The binding process consists of a series of steps taken by the client and server to create, make available, and assemble all the necessary information, followed by the actual RPC, which creates the final binding and routing using the elements established by the previous steps.

**Server Binding Model**

Figure 120 on page 206 and Figure 121 on page 207 show the set of relationships that a server must establish to receive remote procedure calls. As the figure indicates, these are maintained in several places:

- By the server runtime
- In the stub and application code
• By the endpoint mapper
• By a name service

The steps that server applications take to establish these mappings are not discussed here since they are
fully documented in the [z/OS DCE Application Development Guide: Core Components](#). Once
established, this set of relationships allows the server runtime to construct a complete binding, with routing
to a specific server operation, for a call that contains the following information:

• Full or partial binding information
• An interface identifier
• An object UUID, which may be nil
• An operation number

Note that the server runtime itself maintains only a very limited set of relationships: interface identifier/type
UUID/manager EPV and object UUIDs/type UUIDs. It is especially worth noting that the runtime maintains
no relationships between the protocol-address bindings it has created and any of the other information.
The server merely advertises the relationships it wants to export in a name service and registers them in
the endpoint map. Bindings are exported and registered along with interface identifiers and, possibly
object UUIDs.

When the exported and registered information is used by clients to find the server, client calls arriving at
the server endpoints should contain interface identifier/object UUID pairs that the server can, in fact,
service, although the RPC mechanism itself can provide no guarantee of this. This means that name
service and endpoint map operations, while they are not, strictly speaking, a required part of an RPC call,
usually play an important role in constructing bindings.

The indirect mapping from object UUID to type UUID to EPV (and hence to the manager called) also gives
the server some flexibility in organizing its resources based on object UUIDs. This is explained in “Call
Routing” on page 206.

**Client Binding Model**

To make a call, the client needs a compatible binding: that is, one that offers the interface and version
desired, uses a mutually supported protocol sequence, and if requested, is associated with a specific
object UUID.

Clients typically find compatible bindings by making calls to RPC API routines that search the name
service. Typically, the client specifies the interface and object UUIDs desired, and the runtime takes
responsibility for finding bindings with protocol sequences that it can use.

For each binding that the client imports, the runtime provides a server binding handle that refers to the
binding information maintained by the client runtime. This includes the protocol sequence and address
information for the server host and possibly includes an object UUID.

Once the client has found a compatible binding, it makes a call using the binding handle for that binding.
When the call is made, the client runtime has available to it the binding information and any object UUID
referred to by the binding handle. Also available in the stub code are the interface identifier of the
interface for which the call was made, and the operation number of the routine being called. Recall that
the last three items of this tuple of information — the object UUID/interface identifier/operation number —
are precisely what the server needs to route the call to a specific manager operation.
Call Routing

Once the server and client have taken all the necessary steps to set up server and client side relationships, the call mechanism can use them to construct a complete binding and call routing when the call is made. When the client makes a call with a binding that lacks an endpoint, (which is typically the case for bindings imported from the name service) the endpoint is acquired from the endpoint mapper on the target host. The endpoint mapper finds a suitable endpoint by searching the local endpoint map for a binding that provides the requested interface UUID, and if requested, object UUID.

The endpoint map interface and protocol information must match in order for an endpoint to be found, but an object UUID match may not be required. A server can provide a default UUID match by registering the nil UUID. Calls with a nil or unmatched object UUID will get the default endpoint.
Once an endpoint is selected, a call can be routed to one of the endpoints being used by a compatible server instance. The server can unambiguously select the correct interface and operation by using the interface identifier and operation number contained in the call. Recall, however, that the RPC mechanism makes it possible for a server to implement multiple managers for an interface. Hence it may be necessary to select the correct manager. Manager selection is based on the object UUID contained in the call. The selection mechanism depends on two of the relationships established by the server: the object UUID/type UUID mapping and the interface ID/type UUID/manager UUID mapping.
For routing, the server provides a default path by registering a default manager for the nil type UUID. Calls containing the nil object UUID, or any UUID for which the server has not set another type UUID, will be directed to the default manager.

Once the manager is selected, the call is dispatched via the selected manager EPV using the operation number contained in the call.

**Routing Policy**

There are many ways in which clients and servers can arrange the details of binding among themselves, including: how bindings are exported and imported, whether object UUIDs are used, and how object-type mappings are established. High-level resource policy issues relating to the name service and endpoint mapper are discussed in [Chapter 7, “Using the DCE Name Service” on page 215](#). In the present chapter, some of the lower-level routing policy questions that arise from the binding model itself will be discussed. These, in fact, have a substantial impact on how the namespace is used by applications.

The most important issues concern the role of UUIDs in the binding model. Interface identifiers, which consist of a UUID and version number, have a well-defined and unambiguous role. But object UUIDs are somewhat overloaded by the binding model. An object UUID may be used to select bindings from the name service, to select endpoints from the endpoint mapper, and to map a call to the correct manager type within the server. Furthermore, a server may use object UUIDs in some application-specific way to identify and manipulate the objects it manages.

There is great potential for conflict between the use of object UUIDs to select bindings and endpoints and their use to identify objects and routes to manager types. This conflict is particularly evident in the case of servers that provide so-called ubiquitous interfaces, such as the `rdacl` interface. Because many servers on a host are likely to export such an interface, it is essential to have an object UUID to identify the correct endpoint in the endpoint map. Without an object UUID, the endpoint mapper can only return the endpoint of some server that exports the requested interface, very likely the wrong one.

An alternative strategy does exist: a client can call `rpc_ep_resolve_binding()` using a non-ubiquitous interface that it knows the server of interest does export. The call to the ubiquitous interface can then be made with the resolved binding. Clients often use this technique to call the remote `rpc_mgmt` routines. Nevertheless, the objection remains that it is still impossible to select among endpoints of servers or server instances that export the same nonubiquitous interface.

**Object UUIDs as Endpoint Identifiers:** The most straightforward solution is for a server to export a UUID to the namespace where it functions as an unambiguous tag for the servers' endpoints. Clients can find this UUID either by importing it from a named entry or it may be made well-known, effectively becoming a stable, well-known tag for the server's volatile endpoints. When endpoint UUIDs are well-known, they become useful for finding servers even when the client is interested in a nonubiquitous interface. Exactly how servers export and clients find these UUIDs depends on the resource model adopted, as discussed in [Chapter 7, “Using the DCE Name Service” on page 215](#).

This obvious use of UUIDs as endpoint identifiers, however, potentially conflicts with their use as object identifiers. According to the RPC binding model, when clients import bindings based on object UUIDs, these UUIDs are incorporated into call bindings where they may used for endpoint selection, for manager selection, and possibly for some application-specific purpose. If an application exports its object UUIDs to the namespace, then they are used both to identify objects and to identify endpoints. This means that, at a minimum, a server would need to maintain a potentially large number of mappings to the same endpoints.

Moreover, especially when servers manage many objects or create them dynamically, clients will typically know objects by names rather than by UUIDs. Servers can provide such mappings via the namespace
itself by exporting each object UUID to a different namespace entry, but this even further complicates the servers job of maintaining its exports and mappings.

The obvious solution to these problems is to have servers maintain their object UUIDs and name-to-object UUID mappings internally. The basic RPC binding mechanism does not provide much support for this approach: there is no generic way for servers to make objects or names available to clients except through the name service. Also, a UUID used to identify a server endpoint is probably useless for call routing to a manager type within a server. However, the higher-level object management interfaces discussed in Chapter 7, “Using the DCE Name Service” on page 215 provide this functionality.

This leads to two important recommendations:

- Servers should export to the namespace at least one UUID as a tag for its endpoints, and should register the UUID with the endpoint map.
- Servers which support multiple objects should also support the object management interface(s) discussed in Chapter 7, “Using the DCE Name Service” on page 215, instead of exporting multiple object UUIDs to the namespace.

## Binding Handles

Binding handles, although they appear as parameters of RPCs, are in fact purely local to the server or client applications that use them. A binding handle is simply a reference to binding information that is cached by the local runtime. The runtime uses this binding information to construct its side of a client-server association. Even when a binding handle appears as an explicit parameter of an RPC, it is not marshalled or unmarshalled as call data in the same way as other call parameters.

On the client side, a binding handle parameter simply permits an application to indicate explicitly to the runtime which cached binding should be used for the call.

On the server side, a binding handle parameter provides a manager operation with a reference to cached binding information for the calling client so that the manager can, for example, extract authorization information about the client.

In calls to ubiquitous interfaces, such as the `rpc_mgm` interface, partial bindings without an object UUID are rarely adequate, since the endpoint mapper cannot know which server supporting the ubiquitous interface is of interest to the client. The usual model is that the ubiquitous interface is not exported to the namespace. Instead, the client imports bindings based either on another interface supported by the server or an object UUID. If servers follow the recommendation to export at least one UUID with their bindings, no additional preparation will be necessary to allow their clients to successfully call the ubiquitous interfaces they offer. If they do not export the UUID, they will have to adopt the `rpc_ep_resolve_binding()` method described in “Routing Policy” on page 208.

## Binding Methods

In view of what was said earlier about binding handles, the binding method chosen also will be a purely local matter for the client application and stubs. For example, it is perfectly feasible for a server manager to make explicit use of binding information via a binding handle parameter in a remote call, even though the client does not use an explicit handle for the call.

DCE RPC provides the automatic, implicit, and explicit methods for clients to manage bindings for remote procedure calls:

- Automatic method
This is the simplest method of managing the binding for remote procedure calls of an entire interface. With the automatic method, the server exports its binding information to a namespace, and the client stub automatically manages a binding for the application code.

The automatic method completely hides binding management from client application code. The stub imports the binding information and maintains a binding handle. The stub passes the binding handle to the runtime with the remote procedure call, and the runtime uses the binding handle to retrieve the associated binding information. If the client makes a series of remote procedure calls to the same interface, the stub passes the same binding handle with each call.

With the automatic method, a disrupted call can sometimes be automatically rebound. The automatic rebinding requires either that the remote procedure never begins to execute or that the operation is idempotent. If the call meets either of these requirements, the RPC runtime automatically tries to rebind the client to another server (if one is available).

- Implicit method

This is a relatively simple method of managing a binding for an entire interface. With the implicit method, prior to making any remote procedure calls, the client application code obtains server binding information from a namespace or a string binding. The client assigns a server binding handle to a global variable in the client application (for each interface using this method). When calling a remote procedure using the implicit method, the client stub passes the specific interface's global binding handle to the runtime.

**Note:** Multithreaded clients must be careful not to allow one thread to change the value of the shared global binding handle while another thread is using it.

- Explicit Method

This is a more complex yet more flexible method of managing a binding. As with the implicit method, the explicit method requires that the client application code call runtime routines to initialize a binding handle. In the explicit method, however, this binding handle is supplied by the application code as a parameter to the remote procedure call. By allowing a client to manage bindings for individual calls, the explicit method enables clients to meet specialized binding requirements.

The following figure shows the distribution of responsibility for binding management in applications for each of the three methods. The top portion of each box represents the client application code written by the developer. The bottom portion of each box represents the client stub code generated from an IDL interface definition.
You can see from this figure that with the automatic method, binding management belongs completely to the client-stub code generated by the DCE IDL compiler. The implicit method provides the application developer with some control over binding management without having to pass a binding handle as a call argument. With the explicit method, the application developer is completely responsible for binding management. The automatic method requires the server to store binding information in server entries in a namespace; the implicit and explicit methods work with any source of binding information.

A client can use a combination of methods, for an individual interface or more than one interface. For example, one interface might use the automatic method, another interface could use the implicit method, and a third could use the explicit method. In addition, some procedures for the interfaces that use automatic or implicit methods could use the explicit method instead. The method(s) of binding management for an interface is specified using the interface definition, the Attribute Configuration File (ACF), or both. In the interface definition, the explicit method can be specified for the whole interface, or for an operation by declaring a binding handle (using the IDL type `handle_t`) as the first parameter of the operation declaration.

The automatic and implicit methods are interface wide and therefore mutually exclusive; that is, for a given interface, a client can use only one of these interface-wide methods. A client that uses either the automatic or implicit method for an interface can also use the explicit method for some or all of the remote procedure calls to that interface. If the remote procedure call has a binding handle parameter, the explicit method takes precedence over either the automatic or implicit methods of managing bindings.

Explicit and implicit binding both give the client application means to select and modify the binding information used by calls. Explicit binding allows the client to specify binding information per call. This method may be established either by declaring a binding handle parameter as the first parameter for a call in the IDL, or by applying the `explicit_binding` attribute in the associated ACF, either to the interface as a whole, or to specific operations.

Implicit binding allows the client to establish a default binding for an interface. When the `implicit_binding` attribute is applied to a data item in the ACF, then each call that does not specify an
explicit binding parameter (either in the IDL or via the `explicit_binding` attribute in the ACF) uses the default binding information referenced by the implicit binding data item.

With automatic binding, the client stub finds a usable binding for each RPC. Automatic binding is the default for any operation when the following three things are true:

- Implicit or explicit binding has not been specified in the ACF for the interface
- The call does not specify an explicit binding handle parameter
- The ACF does not specify explicit binding for the call

The semantics of automatic binding may differ between the first and subsequent calls on an interface. When the runtime does not have a cached compatible binding, the stub will perform a namespace search to find and import one. The imported binding will be cached for use in subsequent calls. If the client-server connection for the cached binding fails, the client stub will attempt to find a new binding. Therefore, it is possible that later calls will not be made on the same binding, and possibly will even be made to a different server.

A server binding handle that the runtime provides directly to an application is a primitive binding handle. To declare a primitive binding handle, application code uses the predefined RPC binding handle data type `rpc_binding_handle_t`, and an interface definition uses the IDL data type `handle_t`. Primitive binding handles offer a simple means of referring to binding information, which works in most cases. The automatic method of binding management always uses primitive binding handles.

Applications that use the implicit or explicit methods of binding management can choose to store primitive binding handles in an application-specific data structure known as a customized binding handle. Customized binding handles enable application developers to manage binding information to meet the special needs of a specific application. For example, a customized binding handle can be the handle of a file whose records contain the information required to construct a string binding.

Using customized binding handles requires the application developer to perform several special tasks. The RPC interface definition must include a declaration of the customized binding handle as a data structure with a handle data type; this is done by using the `handle` attribute. The client application code must contain specialized procedures that the client stub calls to obtain a primitive binding handle from the customized handle and to release any resources, such as memory, used for the customized handle.

When a customized binding handle is used with the explicit method, responsibility for setting the binding handle shifts to the client stub. The client code provides procedures for obtaining the primitive binding handle from the customized handle and for freeing the primitive binding handle after the call completes. However, it is the stub that calls these procedures to set and free the primitive binding handle.

Calls made with a context handle and no explicit binding handle also have automatic binding semantics. That is, such calls will use the cached binding associated with the context handle. Of course, this binding may have been constructed by the client application and passed, either as an explicit or implicit binding, to the call that returned the context handle. Also, the stub will not attempt to renew such a cached binding if the client-server connection fails. Even if the server is still running and the connection could be reestablished, the server will have rundown the context it is holding for the client, so that the context handle will no longer be valid. When implicit binding is in effect, a call made with a context handle and without an explicit binding parameter will use the cached binding associated with the context handle rather than the implicit binding.

The following table summarizes the binding semantics applied to a client operation:
Authentication and Binding Methods

When a binding handle is selected automatically by the client stub, there is no way for the application to specify authentication data. In principle, it would be possible to have the client authenticate itself to the server in such a case, although a client that does not care about which server it calls obviously cannot authenticate the server. In practice, calls made with automatic bindings are simply unauthenticated. Therefore, if your application cares about authentication, it should avoid using automatic binding.
Chapter 7. Using the DCE Name Service

Correct use of the DCE RPC Name Service Interface (NSI) is essential to the operation of a distributed application, since NSI is the medium through which the application's distributed parts must find each other. NSI works with named database entries which are hierarchically organized into subdirectories and referenced by the familiar pathname convention.

Introduction to Using NSI

It is important to remember that names and objects are separate things in DCE. Consider, for example, these two DCE names:

`/.../tinseltown.org/dce/printers/macmillan`

`/.../tinseltown.org/dce/employees/goethe`

These strings are not file names or file directory names; if you attempt to execute the `ls` command on them, you will only get an error message. They are pathnames that identify entries in the DCE Directory Service, which is DCE's database for storing distributed information. This database is often informally referred to as “the namespace.”

The most important type of distributed information stored in the namespace is information that enables RPC clients to rendezvous with RPC servers; it is called “binding information.” The Directory Service can be used to hold other kinds of data too, but the main subject of the following discussions will be its use as a binding repository.

The set of binding name entries is like a huge data structure of pointers from object names to object locations, and the Directory Service is used mostly as a public DCE locational database, enabling servers to advertise themselves and the objects and resources that they manage, and clients in turn to find and access them. You should never confuse objects with their names; the two are separate things. In particular, the directory service data associated with a name is held in one place (namely, the directory server's database), while the data associated with the object named is held in other place (namely, the object server's database).

How then, you might ask, are file names represented in DCE? Here are two examples of remote file names:

`/.../tinseltown.org/fs/doc/jones/app.gd/chap2.ps`

`/.../tinseltown.org/fs/doc/tolstoy/novels/war_and_peace/chap2.ps`

As you may have guessed, these are also namespace entries, but the entries in this case refer to remote files, and the entry name as a whole is the remote file name. What makes these names different from the other two names given earlier is their third element `fs/` which identifies a “junction” from the DCE Directory Service's namespace into the DCE Distributed File Service's own, separately maintained, namespace. In other words, `/.../tinseltown.org/fs` is the DFS file server's DCE namespace entry. This means that any attempt by a file service client to access a file object whose name begins with `/.../tinseltown.org/fs` will implicitly bind to this server, which will then be responsible for finding, in its own namespace, the file object referred to by `doc/jones/app.gd/chap2.ps` or `doc/tolstoy/novels/war_and_peace/chap2.ps`, and performing the requested operations on it.
The UUID

Thus, it is a mistake to suppose that a name is identical to an object. The name merely points in the direction of the object it names. Objects do, however, have identifiers. These are the 128-bit Universal Unique Identifier (UUID) data structures, which are the identities that the DCE components recognize. They are not usually seen by users, although they play a part in the object-finding process.

UUIDs are used within DCE to identify all sorts of things. From the standpoint of the application programmer, they have two main uses: to identify objects and to identify interfaces.

Object UUIDs

Although “object” is necessarily a rather vague term, a reasonable definition would be the following: an object is any DCE entity that can be accessed by a client, and which can be represented by a namespace entry and identified therein by a UUID. This category can include servers, devices, and other resources. UUIDs that are used in this way are called “object UUIDs” in order to distinguish them from the other main use of UUIDs, namely to identify interfaces (“interface UUIDs”). The difference between these two uses consists only in the way the UUIDs are interpreted by the name service and RPC runtime. Note that it follows from this discussion that an interface is usually not an object. Clients do not normally access an interface as such; the interface is rather a description of the rules of access.

As far as the DCE RPC and name service mechanisms are concerned, it is enough if a client is brought into contact with some server, as long as that server offers the service the client is looking for; in other words, as long as the server offers the interface the client wants to use. To accomplish this rendezvous, interface UUIDs are sufficient. They are also mandatory. There cannot be a client/server relationship without an interface, and the entire RPC runtime mechanism is dependent on the concept of interfaces.

Object UUIDs are different. The RPC runtime usually does not care if they are present or not. But if they are present, they activate various runtime mechanisms that allow clients and servers to be much more specific (always within the bounds of a given interface) about what servers are bound to, and/or what resources the servers will use to fulfill the clients' requests. How this works is explained later in this chapter.

Interface UUIDs

Every IDL-compiled interface specification has its own UUID associated with it. This interface UUID is included by IDL-generated stub routines with every operation request (or return) sent over the network by clients and servers. In this way receiving stubs ensure that they and the sending stubs are sharing exactly the same interface. If the interface UUIDs are different, or are not present, then the remote call will not be completed. But interface UUIDs, although required, play only a secondary role in a client’s finding a server that offers the interface. The main tool for this is NSI, which makes use of the DCE Directory Service, as explained later in this part of the chapter.

Summary: Names and UUIDs

Both names and UUIDs identify objects. But names are separable from the objects they identify, and are only as trustworthy as the binding information their entries contain. UUIDs, on the other hand, are not separable from their objects. Once the desired binding information for an interface or an interface/object combination has been found and used, the name that was used to retrieve it can be forgotten; it is of no further use. This is not true of either interface or object UUIDs.

Note that names become completely unnecessary only if clients have some other means of obtaining valid binding information for the desired service, such as string bindings.
Figure 123 on page 217 illustrates how the information a client finds through a name is turned into network contact with the object named.

![Diagram showing the process of binding to an object](image)

**Figure 123. How a Name Turns into an Object**

**Binding to an Object**

The difference between reading a local file on a single machine and performing the same read on a remote file in DCE is like the difference between reading information from a phone book yourself and dialing an operator for the same information. The remote operation requires the addition of another active entity that can be requested to perform it, since you cannot. Associated with every piece of remote data available on a network is a remote server to manage that data and make it available. The user may not see the server; even the client may be unaware of it, but it is there.

The DCE documentation often speaks of “binding to an object.” In reality, clients can bind only to servers, which then may be requested to perform operations on objects that are under their management. However, it is possible for a server to put bindings into namespace entries that are named for the objects that it manages. Furthermore, these exported bindings can be tagged with object UUIDs in such a way that incoming remote calls from clients can be applied by the server to the object whose name entry the binding was read from (the details of this technique are described later in this chapter). When an application uses this kind of binding model, it is reasonable to say that the client is logically bound to the object, although it is physically always bound to the server that manages the object.

**Junctions**

Namespace junctions are another example of the “hidden server” effect. The following remote file name was discussed earlier:

`/.../tinseltown.org/fs/doc/jones/app.gd/chap2.ps`

and there it was explained that

`doc/jones/app.gd/chap2.ps`

is an entry in DCE DFS's own namespace, while

`/.../tinseltown.org/fs`
is a DCE namespace entry. Suppose a user enters the following:

```
ls -l /.../tinseltown.org/fs/doc/jones/app.gd
```

The clerk agent program (called as a result of the user's entering `ls`) will bind to the remote file server via its `/.../tinseltown.org/fs` DCE namespace entry, and pass to it the residual DFS entry name `doc/jones/app.gd`, along with other parameters. The `ls` command behaves this way because the underlying (VFS+ layer) system calls are coded that way. The DFS server then performs the request (note that the details of interaction within DFS are somewhat more complex than implied by this description). The user only types the command line; the rest is done by DCE, and a directory listing appears on the user's screen.

The VFS+ system routines are used by all possible clients of DFS services (for example, `ls` and `rm` commands, or `fopen()` and `fclose()` library routines). Because of this, and because they know about (and bind to correctly) the remote file server at `/.../tinseltown.org/fs`, the transition from the DCE to the DFS namespace is completely transparent to users. This is how junctions work. As long as all possible clients handle a name that includes a junction correctly, the junction will not be perceptible to the clients' users.

**A Junction Example**

Figure 124 illustrates the principle of junctions. A junction server, which is reached normally through binding information in the DCE namespace, maintains its own namespace of named objects. The junction server's clients allow users to refer to these objects by actually concatenating the server's entry name and an object's “internal” name. The client then breaks this string apart by contacting the server named in the first part of the string, and passing to it the second part, which is a valid name within the server's namespace. The client's user seems to access the object directly.

![Diagram of a namespace junction](image)

Figure 124. A Namespace Junction

The dashed lines in the above figure show the progress of the Client's efforts to get access to the desired Object, which involves acquiring a binding to the Junction Server, making contact with it, and passing to it the Object's Name. The solid line shows the apparent direct access to the Object that the Client's user seems to enjoy. The dotted lines show other possible paths of access to the other Objects that the Server manages.

Junction protocol is generally a private matter between an application's clients and servers. However, the `acl_edit` command uses a generalized protocol.
Junctions and the ACL Editor

The binding routines that acl_edit uses are discriminating enough to detect a junction anywhere in an entry name that is passed to it. This allows a distributed application to have its own namespace for objects with ACLs on them, rather than burdening the DCE namespace by separately exporting binding information for every one of these objects. The separate objects have to be made publicly accessible somehow because entities should be able to access ACLs directly, regardless of whether they happen to already be in contact with the server that manages the ACLed object, and indeed regardless of whether or not they happen to be a client of the particular server to which the objects belong.

Suppose, for example, a user enters

```
  acl_edit /.../tinseltown.org/dce/dce_print/cotta
```

in order to interactively edit the ACL for the printer object cott, where /.../tinseltown.org/dce/dce_print is the namespace entry for a print server, and there is no /.../tinseltown.org/dce/dce_print/cotta entry in the DCE namespace. The binding routine, sec_acl_bind(), which is called internally by acl_edit, receives an error when it tries to bind to the object cott. However, the DCE Directory Service also tells it how much of the name it passed is valid. The sec_acl_bind() routine then retries the binding operation, this time through the valid entry name (/.../tinseltown.org/dce/dce_print), and passes the residual part of the name (cotta) as a parameter. Now it is up to the application ACL manager to interpret the residual name correctly and find the requested ACL.

Name Service Terminology

DCE RPC NSI is an RPC-based interface that uses the DCE Cell Directory Service (CDS) as its database. The NSI routines do not constitute a general interface into CDS as such; they are a set of specialized routines whose purpose is simply to provide ways for RPC servers to advertise themselves to RPC clients, and for clients to find and bind to them.

In fact there is no public general API (Application Programming Interface) to CDS. There is a general CDS interface that is used internally by the DCE components, but applications normally access CDS through NSI. Applications can get full access to CDS, if necessary, by using the XDS interface.

CDS Entries

NSI uses a subset of the many possible kinds of CDS entries in order to accomplish its tasks. CDS entries are characterized by the CDS attributes they have; each entry can have one or more such attributes. Each separate attribute defines that entry's ability to contain one or more items of a particular kind of simple or complex information.

The name service creates and uses CDS entries that use only the following four attributes:

- **binding** The entry has a field that can contain one or more sets of binding information. When the field is read, a binding handle that contains the necessary information from one of these sets is returned, in no particular order.

- **object** The entry has a field that can contain one or more object UUIDs. When the field is read, one of the UUIDs is returned, in no particular order.

- **group** The entry has a field that can contain a pool of one or more references to other (independently existing) NSI entries; each time the field is read, one of these entries is returned. Different entries are returned on successive reads, but the order of return is undefined. Note that the “other NSI entries” referred to in the group can themselves be server or group entries. As a result, the act of reading from a group attribute can, depending on the actual API routine called,
lead to a series of nested operations. Any nesting is transparent to the client application, which seems to perform a simple read and to receive the contents of a single entry in return.

**profile** The entry has a field that can contain one or more prioritized elements, each of which consists of a reference to another (independently existing) NSI entry. When the field is read, the elements are read in a specified order. The entry referred to in the element may itself be a server or a group or a profile. As a result, any element may in fact, depending on the actual API routine called, resolve on access to a nested path of referred-to entries. As with group entries, this is transparent to the client application.

Although a single entry could contain both group and profile attributes (and for that matter, binding and object attributes as well), it is not a good idea to mix attributes in this way because the results of importing (reading) from such an entry are too indeterminate.

The typical name service entries are as follows:

**server entry** Contains a binding and an object attribute, making it suitable for containing the necessary binding information for a single server.

**group entry** Contains a group attribute.

**profile entry** Contains a profile attribute.

There are no official names for hybrid entries that contain other combinations of attributes, which is perhaps another reason for not creating such entries.

The general name for entries that contain any of these attributes is “NSI entries,” since they are a by-product and tool of the NSI DCE RPC library routines.

**CDS Entry Attributes**

Within the DCE Directory Service, entry attributes such as the four previously described attributes are identified by Object Identifiers (OIDs). This is an exception to the general rule that things in DCE are identified by UUID.

OIDs are not seen by applications that restrict themselves to using only the name service routines (rpc_ns_...()), but these identifiers are important for applications that use the X/Open Directory Services (XDS) interface to create new attributes for use with namespace entries.

As was seen in the immediately preceding sections, the name service makes use of only four different entry attributes in various application-specified or administrator-specified combinations. CDS, however, contains definitions for many more than these, and attributes from this supply of already existing ones can be added by applications to NSI entries through the XDS interface. Attributes that already exist are already properly identified, so applications that use these attributes do not have to concern themselves with the OIDs, except to the extent of making sure that they handle them properly.

A further possibility is that an application requires new attributes for use with namespace entries. Such attributes can be created using the XDS interface. When it creates new attributes, the application is responsible for tagging them with new, properly allocated OIDs.

Unlike UUIDs, OIDs are not generated by command or function call. They originate from the International Organization for Standardization (ISO), which allocates them in hierarchically organized blocks to recipients. Each recipient (typically an organization of some kind) is then responsible for ensuring that the OIDs it received are used uniquely.

For example, the OID
1.3.22.1.1.4 identifies the NSI profile entry attribute. This number was assigned by the Open Software Foundation out of a block of numbers, beginning with the digits 1.3.22, which was allocated to it by ISO, and OSF is responsible for making sure that 1.3.22.1.1.4 is not used to identify any other attribute.

When applications have occasion to handle OIDs, they do so directly, since the numbers do not change and should not be reused. However, for users’ convenience, CDS also maintains a file (called `/opt/dcelocal/etc/cds_attributes`) that lists string equivalents for all the OIDs in use in a cell, in entries like the following one:

```
1.3.22.1.1.4   RPC_Profile byte
```

This allows users to see RPC_Profile in output, rather than the mysterious 1.3.22.1.1.4. Further details about the `cds_attributes` file and OIDs can be found in the “CDS Attributes File” section in the Z/OS DCE Administration Guide.

Broadly speaking, the procedure you should follow to create new attributes on CDS entries consists of three steps:

1. Request and receive, from your locally designated authority, OIDs for the attributes you intend to create.
2. Update the `cds_attributes` file with the new attributes’ OIDs and labels; that is, if you want your application to be able to use string name representations for OIDs in output.
3. Using XDS, write the routines to create, add, and access the attributes.

Non-NSI attributes on NSI entries can be very useful, even though you cannot access the extra attributes through the name service routines but must use XDS instead.

### Binding

In order to highlight the essentials of name lookup and storage and the management of binding information, many details of DCE RPC operation are either greatly simplified in the following descriptions or omitted altogether.

A binding is a package of information that describes how a client can contact and communicate with a particular server. Although the underlying protocol that implements the communication can be connectionless or connection-oriented, the relationship itself is still expressed as a binding.

### Importing and Exporting Bindings

The name service exists to store server binding information into the cell namespace, and to retrieve that information for clients. Using NSI, servers export their binding information to be stored under meaningful names, and clients import these bindings by looking up those names. Thus, the locations of the servers can change, but clients can continue to use the same names to get bindings to the servers. Figure 125 on page 222 shows how client and server use the name service.
When a prospective client attempts to import binding information from a namespace entry that it looks up by name, the binding is checked by NSI for compatibility with the client. This is done by comparing interface UUIDs. The client presents an interface UUID when it begins the binding import operation; the UUID of the interface being offered is exported to the name entry, but not in the binding handle itself, by the server. If these interface UUIDs match, then the binding handle contained in the entry is considered compatible by the RPC runtime and is returned to the client. If more than one handle is contained in the entry (this is often the case), they are returned one by one on successive imports. NSI also checks for protocol compatibility.

The import routines will return only client-compatible bindings, but a client can sift through the returned bindings and make its own choice as to which ones to use, based on its own criteria. The technique by which this is done consists of converting the bindings into string bindings, and then inspecting (or comparing) the strings.

Note that binding handles do not include an interface UUID. Binding handles do contain a host address, an endpoint, and an optional object UUID, among other things. The interface UUID is associated with the interface's stub code, which inserts it into outgoing RPCs and checks it in incoming ones, thus guaranteeing client/server operational compatibility. This allows binding handles to be used very flexibly. Once a client has successfully bound to a server, it can utilize any of the interfaces that server offers, simply by making the desired remote call.

**Summary**

The mapping from name to server that occurs when bindings are imported from the namespace is indirect because binding is a two-step process. First, the binding handle is obtained by lookup from a named entry, and then the handle is used to reach a server. The crucial point is that the imported handle will not usually contain a complete binding to a specific server (namely, the one that happened to export it). Completion of the partial binding occurs later, when the client makes its first remote procedure call; the RPC runtime uses UUIDs, not names, to determine how it should complete a binding.
Partial Binding and the Endpoint Mapper

Binding handles imported by clients from the namespace normally contain only partial binding information. The exported binding information is sufficient to locate the DCE Host daemon on the server’s host (the machine the server resides on), but it does not yet include a specific endpoint (UDP or TCP port number) for the desired service on that host.

The reason for omitting dynamic endpoint information in exported binding handles is to avoid unnecessary multiplication of accesses to the namespace. Since dynamically generated endpoints are reassigned every time a server starts up, entering them into the namespace (and thus forcing CDS to propagate the new information throughout the various directory replicas) would greatly increase namespace housekeeping chores.

Thus, the last step in the binding process is obtaining an endpoint. The step is performed transparently as far as the client is concerned. It is accomplished by the endpoint mapper service of the DCE Host daemon, dced, when the client makes its first call to the partially bound-to server. The endpoint mapper service manages its own private database of server endpoints for the host on which it is located. The endpoints are registered by the servers as part of their startup routine.

The binding information that accompanies a prospective client’s first remote procedure call takes that call to the well-known endpoint of dced on the exporting server’s host machine. The endpoint mapper now takes over. It looks up a valid endpoint for the requested service, copies it into the binding handle, and transfers the call to that endpoint. Subsequent calls from the client, which now has a binding with one of the server’s endpoints, will bypass the endpoint mapper.

The endpoint mapper picks an appropriate endpoint for an incoming partial binding by matching interface UUIDs by default. Any endpoint that has been registered under an interface UUID that matches the incoming interface UUID, which identifies the interface requested by the prospective client, is eligible for selection. This mapping process is called “forwarding” when it occurs with connectionless protocols, and “mapping” when it occurs with connection-oriented protocols.

Figure 126 shows the endpoint mapper service completing a binding.

![Diagram of endpoint mapper service](image)

**Figure 126. The Endpoint Mapper Service Completes a Binding**

There is an exception to this scheme. Some servers are designed to occupy well-known addresses. The DCE host daemon itself, dced, is reached in this way, making its accessibility independent of whether or
not the namespace is accessible. The endpoint(s) of a well-known address do not change; they are usually specified in the application's interface specification (contained in its .idl file). Bindings to servers that use well-known endpoints are already complete at the time of import; the endpoint mapper never sees these bindings.

**Interface Ambiguity and Partial Bindings**

The interface UUID, which was generated by the IDL compiler, uniquely identifies the set of operations that the client will access through that interface. In short, it identifies the interface. An interface UUID may also happen to identify a server which offers that interface. But if more than one server on the same host offers the same interface (which could easily be the case), the interface UUID alone will not be sufficient to identify a specific server. The result is that if a remote call comes in with such an ambiguous interface and a partial binding, the endpoint mapper will have to randomly choose any one of its eligible registered endpoints, complete the binding with it, and send the call on to that server.

Imagine several print servers residing on the same machine (see Figure 127). Each server manages a group of printers that share a common physical location. All the printers in room “A” are managed by the “A” print server, all the printers in room “B” by the “B” print server, and so on. Now suppose each of these servers has a separate entry in the namespace. Figure 127 shows the sequence of events that occurs.

![Diagram](image)

**Figure 127. Print Server Entries in Namespace**

The following steps describe the sequence of events shown in the above figure:

1. The Client imports a partial binding to the *Printer* interface from the entry “A” in the Namespace.
2. The Client makes its first call with the binding it imported from “A.”
3. The Endpoint Mapper at Print Server A’s host, when it receives the call from the Client, has no way of knowing which of the four Print Servers it should map the call to, since all four servers have registered their endpoints under the same interface. It therefore picks one at random to complete the binding.

The entry names are different, but the partial binding information contained in the entries is identical, since the servers’ host machine is the same. The interface UUID included in the call is no help, since that same
interface is offered by all the servers. A client seeking a print server may not care to which server (and thus to which printer) its request goes, but then again, it may care. If it does, there is a way it can specify a server so that the endpoint mapper can select an appropriate endpoint to complete the partial binding.

Using Object UUIDs to Avoid Binding Ambiguity

Binding handles can contain, besides host address and endpoint information, an object UUID as well. The endpoint mapper will try to match an object UUID contained in a binding handle with one of the object UUIDs associated with its map of registered endpoints. This allows even a partial binding to specify a target more precisely than just by host machine. Since object UUIDs are generated by the `uuid_create()` function call (see the [z/OS DCE Application Development Reference](#)), servers can create as many of them as they need.

For the print server example discussed in the previous section, the namespace entries for the servers could be set up as shown in Figure 128.

---

**Figure 128. Print Server Name Entries with Object UUIDs**

The following steps describe the sequence of events shown in the preceding figure:

1. The Client imports a partial binding to the `Printer` interface from the entry “A” in the Namespace.
2. The Client makes its first call with the binding it imported from “A.”
3. This time the Endpoint Mapper at Print Server A's host is able to match the call with A's registered endpoints, because the endpoints have been registered with both the `Printer` interface and Print Server A's Object UUID, and the incoming call's partial binding also contains Print Server A's Object UUID.
Each server has exported a set of partial bindings that differs from all other servers' by its object UUID (which thus becomes, in effect, a server ID). If, for example, Server A has properly registered its endpoints with the same object UUID as the one it exported its bindings with, the Endpoint Mapper will make sure that a partial binding exported from Server A’s name entry will result in a full binding to Server A.

Now suppose that each print server sets up a separate namespace entry for each printer it manages. The printers themselves would, in effect, be identified by their own object UUIDs. Figure 129 illustrates this.

![Figure 129. Separate Printer Name Entries](image)

Now a client will be able to access a specific printer by importing a binding handle from that printer's name entry. The endpoint mapper at the target host would compare the object UUID in the partial binding with the object UUIDs registered by the print servers, and select an appropriate server. The server in turn would also use the object UUID to select the correct printer for the request, if it managed more than one printer. A namespace set up in this way with a separate entry that contains a unique object UUID for each accessible service resource is called an “object-oriented” namespace.

**An Object-Oriented Namespace**

“Object-specific entries” are namespace entries that each contain binding information only for one specific object or resource, as demonstrated in the last printer service shown in the last previous figure. “Object” can mean any of several things, depending on what kind of service the application's servers are offering. Here are some examples.
Thus, for a client that wants to have a file printed, it is natural to allow it to specify a printer as a destination. Therefore, the client would bind to the print server through a name entry that specifies a printer. To send something to a different printer, the client would import a binding from the name entry for that other printer. The server may (or may not) be identical, but the object UUID in the binding handle returned would uniquely specify the one printer represented by that entry.

On the other hand, consider an application that returns statistics about the processes currently active on a group of machines. In this case it would be reasonable to regard the server as the object. In the namespace entries for such an application, each entry would uniquely represent one server. A client would import a binding from the name entry for the server it wanted to work with.

In other words, "object" is a handy way of saying "the thing that clients will want to access" in order to accomplish the task set for the application. If the namespace is organized correctly, clients will be able to import bindings from these objects' entries.

### Setting Up an Object-Oriented Namespace

Once you have distinguished the objects your application uses, you must decide on an appropriate set of names for the entries themselves. The entries can be created either by the application (server), if it has the necessary privileges, or by a system administrator using the `rpccp` command interface.

After the entries have been created, each server must do the following:

1. Create an object UUID for each object managed by the server under an interface, insert it into the binding handle(s) for that object, and export the handle(s) for each object to a separate entry in the namespace.

   Note that the object UUID should be generated and exported in general only once per created namespace entry, and not each time the server starts up (see the example that follows of how to do this). When a newly restarted server exports its partial bindings, nothing actually happens in the namespace because the partial binding information remains the same (unless the server has moved to a different machine). However, if the object UUIDs are regenerated, then the change in exported information will force needless update activity in CDS, which is where the entries exist.

2. Register with the endpoint mapper the full bindings (including endpoints) obtained for the interface; `rpc_ep_register()` performs this operation.

One way of avoiding unnecessary regeneration of object UUIDs would be to have a restarted server check the namespace for the presence of its previously exported object UUIDs, as demonstrated in the following code fragment. Refer to the [z/OS DCE Application Development Reference](#) for further information on the function calls.

```c
have_object = false;

/* Create an inquiry context for inspecting the object */
/* UUIDs exported to "my_entry_name"... */
rpc_ns_entry_object_inq_begin(my_entry_name_syntax, my_entry_name, &context, &st);
```
Whenever you want to offer more than one instance of the same interface on the same host, you must distinguish by object UUID the binding information in the name entries exported by the servers, if it is important to distinguish among the servers when binding to them. Otherwise, the endpoint mapper's selection of an endpoint with which to complete the binding from among all the servers on that host that offer the appropriate interface will be random.

Figure 130 on page 229 illustrates what such an object-oriented namespace should look like.
Each entry has a name denoting the object represented, although the names are not shown in this figure.

Under this model, clients bind to servers via named objects in the namespace, each of which contains enough specific information in its partial binding to allow the endpoint mapper at the destination host to choose an appropriate endpoint for the incoming RPC.

By setting a namespace up this way, however, you do not necessarily restrict yourself to this one model for accessing binding information. Through the use of two other types of entry, groups and profiles, which can be superimposed on the simple object model, you can set up models where clients bind to abstractions such as services, or directly to the servers themselves. These techniques are described in the next section.

Nevertheless, at this point you have enough information to set up a namespace that consists of an entirely “flat” expanse of separate resource entries. Bindings can be imported by clients by looking up specific names. If the client has no specific name to look up, or if the lookup on the name(s) it has fails, it has no alternative way of binding to a server.

**Groups and Profiles**

Name lookups can be made more flexible with two other types of entry: groups and profiles.
Group Entries

A group entry consists essentially of multiple independent other entries whose names are also associated under the group name. These “other” entries can be simple (single-name) entries, or they may themselves be group entries. Doing an import from the group entry will return the contents (the binding handles) of its included entries (which are called “members”), but the selection is made by the DCE RPC runtime, and from the client’s point of view is undefined and implementation dependent.

In practice, the way this works with the usual binding import operations is as follows. Clients normally import bindings by first calling `rpc_ns_binding_import_begin()` to set up an import context. Once this is done, successive calls to `rpc_ns_binding_import_next()` will return binding handles from namespace entries until the handles have all been returned or the client decides to stop; the client decides which handle(s) to use based on its own criteria. When it is finished importing, it calls `rpc_ns_binding_import_done()` to free the context.

The kind of entry the information is returned from is usually unknown to the client, which needs to know only a name to look up and the interface UUID by which it wants to bind. If the name is that of a simple server entry, then the bindings contained in that entry only will be returned. If the name is of a group entry, then bindings will be returned from members (single entries) of the group, selected (by the RPC runtime) in an undefined order. If one or more members of the group are themselves groups, then the same thing happens recursively whenever these lower-level groups are accessed.

Note that the group entry and its members are separate things. The group entry can be deleted, but its former members will continue to exist as independent entries, unless they too are explicitly deleted. Thus, you can implement a namespace organization where the same bindings can be imported through individual simple entries or through group entries, depending on how the client is coded.

Profiles

A profile entry specifies a search path or hierarchy of search paths to be followed through the namespace in order to obtain a binding to a server that offers a specified interface.

When a client imports from an entry that happens to be a profile, successive imports (accomplished by calling `rpc_ns_binding_import_next()`) return the contents of entries that are read as a result of following the specified path through the namespace. All this is transparent to the client, which sees only the bindings returned. Profiles can be used to set up default paths and groups of paths for users. The `RPC_DEFAULT_ENTRY_NAME` environment variable, which is the default entry name used by the name service in import operations, usually contains the name of a profile.

As with groups, the entries contained in profiles, which are called “elements,” exist independently of the profile entry itself.

A very important property of profiles is that they allow clients to know little or nothing about the organization of the namespace itself. Using the default case as an example, consider the following: if the profile at `RPC_DEFAULT_ENTRY_NAME` has been set up with elements containing entries for all possible active servers for a particular application, then clients can simply import from this name and trust the profile mechanism to walk through the various compatible possibilities and return binding handles via successive calls to `rpc_ns_binding_import_next()`. (Note that a profile entry is not limited to containing entries for just one interface; thus, `RPC_DEFAULT_ENTRY_NAME` could be set up to contain all the defaults for a cell.)
Summary of Namespace Entry Types

Clients access binding information in the namespace by looking up (by name) one of three different kinds of entry:

- A server entry
- A group entry, which contains other entries whose contents are returned to the caller when it reads the group entry
- A profile entry, which specifies a path of entries to be searched whose contents are returned to the caller when it reads the profile entry

Lookups behave differently depending on the kind of entry read. If an entry is a simple server entry, then the search begins and ends right there, whether successful or not. If the entry is a group, then the lookup is more complicated. A binding will be returned from among those that are found to be compatible by the name service, but within that category the selection is undefined. If the entry is a profile, then a specified path of entries is searched. The entries in this path may themselves be other profiles, or groups, or simple entries. The search continues until either a compatible binding is found, or the entire path has been unsuccessfully traversed.

Three Models for Accessing Binding Information

By adding groups and profiles to the object-specific namespace organization originally described, you can implement any or all of the following three basic models for accessing binding information:

- Clients bind to services
- Clients bind to servers
- Clients bind to resources or objects

Each of the three models is described in the following sections.

Access by Services

Servers have separate namespace entries; each server distinguishes the bindings it exports with its own identifier; that is, an object UUID that it generates for itself the first time it starts up. These separate server entries are also members of group namespace entries, which represent services. The criterion for membership in a service group is that all the servers in it export the interface that identifies that service. (They may happen to export other interfaces as well.)

Clients, in effect, bind to services by importing their binding handles from the group entries. Note, however, that the server-specific entries still exist independently and are accessible to lookup.

This model is appropriate for applications where clients do not care which server they happen to bind to or where that server is located as long as it offers the desired service. The eligible servers are pooled into a group entry from which bindings to one of them are selected in an undefined order and returned whenever a client performs an import operation from the group entry.
Access By Servers

In this model, distinct servers have separate and distinct name entries, and clients import bindings directly from the server entries. Hence, an application using this kind of binding model will "own" just as many simple entries in the namespace as there are active servers.

Since the client in this model is looking for a specific server, imports will be done directly from the server entries. The only exception to this rule would be where two or more instances of a server were active on the same host, and it was indifferent to the client as to which one it is bound to. The entries for the multiple same-host servers then could be put into a group entry, and binding imports done from the group.

Access By Objects

Servers operate on or manage multiple objects. Clients use these objects (via the servers) as resources. For each such resource, the server creates a separate namespace entry and exports its binding information there, distinguishing each object entry with its (the object's) own object UUID.

An example of this model is the printer service that was previously described. Clients will import directly from the name entry of the resource they want to use. For this kind of application, there will generally be more namespace entries than active servers, since each server presumably manages more than one object. If the name entries have been set up correctly and the servers have properly registered the object UUIDs they created, there will be no difficulty in routing any partial binding to the correct server (namely, the server that manages the object or resource specified).

Summary of Binding Models

Although the name service allows other approaches, we recommend that whenever possible you use the object-oriented scheme to organize your namespace entries. There are at least two good reasons for doing so. First, it is easy to administer; at the simple entry level, things really are simple. Second, this is the most flexible foundation for building other more complicated access models using group entries and profiles.

The separate name entries in your namespace should contain bindings that will unambiguously resolve to specific server instances. Since interface UUIDs are often offered by more than one server, more information than just an interface UUID is needed in order to give an RPC with a partial binding the required specificity. Object UUIDs provide this extra information. When using object UUIDs to distinguish bindings in this way, servers must take care to preserve their uniqueness across name entries.

Finally, profile entries allow clients to walk through a specified search path of namespace entries and yet be completely ignorant of the actual names themselves. While name independence may not be desirable for an object-based or resource-based distributed application, it can be a powerful mechanism when used with other models.

As you are setting up the namespace organization for your application, remember that there is not a direct exact mapping from names to bound servers. Different names, once imported, may resolve to identical bindings if the partial bindings were exported on the same interface, from the same host, and not otherwise distinguished from each other by object UUIDs. It is the application developer's responsibility to tailor an application's export and import procedures so that this mapping behaves as intended.
Models Based on Non-CDS Databases

The three models previously described are not mutually exclusive; if the namespace is set up correctly, all three can coexist at the same time. All three of the models are implemented through the functionality of the DCE RPC name service.

Although the emphasis in this discussion has been placed on the storage and retrieval of binding information, the namespace entries can be used to store additional states for objects. In order to do this, an application would have to create additional attributes on the CDS entries it intended to use because the name service recognizes only the four NSI attributes: binding, object, group, and profile.

Such additional entry attributes would be created and accessed through XDS. However, whenever you find yourself contemplating extending the name service in this manner, you should carefully consider whether the name service (and, consequently, CDS) is the best mechanism for doing what you want to do.

In the preceding example, where an object-oriented namespace containing separate entries for individual printers was described, only the identifier for the printer (the object UUID) and the binding for the server that managed it were stored in the CDS entry. Other information, such as what jobs are currently queued for the printer, who owns the jobs, and so on, was maintained by the server. This data could be stored in CDS only by creating new attributes to put it in, but it would be changing too quickly for CDS to efficiently keep up with it anyway. The performance of both the application and CDS would suffer from such an arrangement.

It is possible to imagine distributed applications whose resources (the objects they are managing) are of such a nature that they could be more efficiently managed through a private application-implemented database. Suppose the number of managed objects is very large, or that the state of the objects is volatile. It would certainly be a bad idea to try to use CDS to store this kind of information, which would be changing much more rapidly than CDS’s ability to propagate the updates.

Example of a Privately Managed Database

As an example of such a privately managed database, consider a print service where jobs are submitted not to individual printers, but rather to a generic printer service. The client, lpr, binds (probably through a group entry) to some certain print server, and sends the job to be printed to that server, which then, after some thought, sends the job to one of the printers that it manages.

Consider, for example, what happens if a user invokes the client cancel sometime later to stop a job. If, for example, the original command was

```
lpr War_and_Peace.ps
```

and the subsequent request to cancel is

```
cancel War_and_Peace.ps
```

then how does the server that cancel binds to find the right job to delete? There is no guarantee that cancel will bind to the same server that happened to receive the original print request, so having each print server keep track of its own jobs would not be the answer.

One way to keep track of jobs queued would be to have a dedicated “job location server” as part of the application. Each time a print server queued a job to a printer it would record the fact (with all the pertinent details) with the location server. Whenever a job completed, the server would again notify the location server to remove its record of that job from its database. A client cancel then binds first to the location service, where it receives the name of the print server associated with the job it wants to cancel.
It then looks up that name, binds to the right print server, and sends the cancel request. In effect, the location server has become a name service for cancel.

This method of organizing activity results in a split-model database. The print servers' binding information is managed through CDS, as usual, and the location server manages other more volatile information associated with those same servers.

Another way a server could maintain its own database of named objects would be by implementing a junction.

Combining Models

In designing a binding access model for an application, consider also whether it may be appropriate to combine some of the models previously discussed. In the print service application, it may be desirable for servers to also offer a management interface to specific servers rather than to specific objects; for example, lpr, lpq, and lprm are generic application clients, so it is appropriate for them to bind to printer objects, but if lpr_mgmt is supposed to manage characteristics of a whole service, then it should bind to servers.

An Object-Oriented Model with Grouped Binding Information

The following variation on the object-oriented binding model shows how the group attribute can be used in object entries. In this model, each of the object entries contains, as before, an object UUID that will uniquely identify (either to the endpoint mapper on the exporting server's machine, and/or to the server itself) the object referred to by that entry. However, the object entries do not contain any binding information. Instead, a group attribute in each object entry refers clients' import operations back to the server's own separate entry, which contains the binding information for that server.

The namespace ingredients of this model are the following:

- A single namespace entry for the server, which contains a binding attribute and, possibly, an object attribute. Thus, this entry contains all the binding information that is exported to the namespace by the server.
- One namespace entry for each object that the server offers. Each entry contains an object attribute that contains that object's UUID, and a group attribute that refers back to the exporting server's namespace entry.

Note that the object entries consist of a combination of attributes not encountered before (object and group). Although unorthodox combinations of attributes are not generally recommended, they can sometimes be useful, as in this example.

The advantages of this scheme are two-fold:

- It greatly reduces the amount of server-provoked export activity into the namespace.
- It allows the server application to associate a people-readable name (that is, the name of each object's namespace entry) with a UUID.

When the server is first activated it creates all the namespace entries, exports the objects' UUIDs into the object entries, and initializes the group attributes to refer to the server entry. It exports its binding information into the server entry only. From then on, whenever it is restarted, all the server needs to do is re-export its binding information into the single server entry. Everything else remains the same; that is, the objects' UUIDs have not changed, nor has the name of the server entry to which the object entries'
group attributes refer. Thus, instead of exporting bindings to every one of its object entries on subsequent startups, the server exports to only one entry.

Of course, if the system were restarted or the namespace reinitialized, then the original start-up process would have to be repeated.

The slight disadvantage of this scheme occurs on the client side, where the import process becomes somewhat more complicated than it would be if all necessary information (both binding and object UUID) could be read in from the same entry.

---

**Server and Client Steps**

The following subsections describe in detail, from both the server's and the client's side, how this model works.

**Server Export**

This section lists the steps that the server must perform to set up and initialize its namespace. Each step consists of the NSI function that must be called to perform the operation.

1. **uuid_create()**
   - To create an object UUID for each object that the server intends to export.

2. **rpc_server_register_if()**
   - To register interface(s) and EPVs with the RPC runtime. (This is also where manager types, if any, are registered.)

3. **rpc_server_use_all_protseqs()**
   - To request bindings from the RPC runtime for each object.

4. **rpc_server_inq_bindings()**
   - To get the binding handles for each object.

5. **rpc_ns_binding_export()**
   - To export the binding information of the objects' common server and the object UUIDs for each of the namespace objects to the server's own separate name entry. This step is performed **only once** for each collection of objects managed by the same server.

The final three steps set up the grouped collection of service objects. Note that the next two steps are executed once for each object managed by the server:

6. **rpc_ns_binding_export()**
   - To export each object's object UUID to its own name entry. A **NULL** is passed as the **binding_vec** parameter to specify that only an object UUID, and no bindings are being exported.

   Note that each object UUID must be exported to both the object name entry and the server entry; hence the need for this export operation in addition to the operation described in Step 5 above.

7. **rpc_ns_group_mbr_add()**
   - To add the server's name entry (created in the first step) as the sole member of an NSI group attribute in each of the separate objects' name entries created in the second step.
8. **rpc_ep_register()**

To register each object's UUID with the server's host machine's endpoint mapper. Note that **rpc_ep_register()** takes an object UUID vector as an argument, and generates from this all the necessary relationships between UUIDs and bindings; thus the call is made only once.

The point of this step is to make sure that when presented with an object UUID in an incoming RPC, the endpoint mapper can look that UUID up in its database and find an endpoint that has been registered with it. Registering the server's bindings (that is, endpoints) with all object UUIDs will accomplish this.

Step 6 is made necessary by the way the ACL editor's binding mechanism works. (Applications gain access to the ACLs that an application maintains on its objects through the client agent **acl_edit**, which uses a standard DCE-wide interface for ACL operations.) The **acl_edit** mechanism contains code that allows it to bind to the server that implements the ACL manager responsible for the object whose ACL is desired. However, these generalized binding routines conform to certain fixed ways of doing things. If the **acl_edit** binding mechanism obtains an exported object's object UUID from the object entry, it will use that object UUID in its subsequent import through the group attribute.

Thus, the object UUID will be contained in the handle structure that the client presents to the **rpc_ns_binding_import_next()** call, expecting it to be filled in with binding information. However, the RPC runtime always tries to match such an input object UUID with a UUID contained in the entry that the caller is trying to import from. If no matching object UUID is found, no binding information will be returned. Thus, all the single object UUIDs separately exported to the object entries must be exported to the server entry as well, if the exported objects are to have ACLs accessible through the **acl_edit** mechanism.

Figure 131 illustrates the resulting namespace arrangement.

![Figure 131. The Export Operation in a Model with Grouped Bindings](image)

This generic server manages four objects, called simply "A," "B," "C," and "D." One entry is created for each of these objects, and a separate entry is created for the server itself, where the binding information is held.

The result of all this is that there is now one more namespace entry for a given service instance than there would have been with the object-oriented model discussed earlier. The group attribute in each entry is a level of indirection that allows the server to dispense with exporting many copies of the same thing.
If a directory with the proper permissions has been set up for it in the namespace by the system administrator, a server should be able to create the object entries simply by making the calls described here.

**Client Import**

To bind to an object managed by the server as previously described, a client performs the following series of library calls:

1. `rpc_ns_entry_object_inq_begin()`
   To set up an object inquiry context; the client application here specifies the name of the desired namespace object entry.

2. `rpc_ns_entry_object_inq_next()`
   To return the object UUID that the server exported to the object's entry.
   
   This UUID (which will be passed to the `rpc_ns_binding_import_begin()` routine, below) will enable the server host's endpoint mapper to accurately map the incoming remote procedure call to the server that exported this entry.
   
   The UUID may also be used by the server itself to determine which object the client wants to access. Note that although this set of library routines is designed to accommodate schemes in which multiple object UUIDs have been exported to the same entry, the model described here requires that only one object UUID (the unique identifier of the object to bind to) be exported.

3. `rpc_ns_entry_object_inq_done()`
   To delete the object inquiry context.

4. `rpc_ns_binding_import_begin()`
   To set up a binding import context.
   
   Note that the object UUID that was returned by the call to `rpc_ns_entry_object_inq_next()` must be passed to `rpc_ns_binding_import_begin()`: as a result of this the import operation (`rpc_ns_binding_import_next()`) will return only a binding with that object UUID.

   An alternative to using the binding import routines would be to use the group member inquiry (`rpc_ns_group_mbr_inq_...()`) routines to learn the name of the entry referred to in the group attribute, and then to do a direct import from that entry.

   The reason for using the `rpc_ns_group_mbr_inq_...()` routines, rather than the normal import functions (`rpc_ns_binding_...()`, would be to make sure that the group (and not some other) attribute in the entry is read. The `rpc_ns_binding_import_next()` routine is defined to successively exhaust the contents of an entry's
   
   - `binding` attribute
   - `group` attribute
   - `profile` attribute

   Since the model described here employs object entries with only group attributes and no binding or profile attributes, using the normal import routine should work fine.

5. `rpc_ns_binding_import_next()`
   To read the entry's group attribute.
   
   The name service's access to (and return of the binding handle from) the entry's group attribute is transparent and unerring because there is only one set of binding information associated with a given entry in this scheme, and that information is found only in the group attribute. Note that if there had been more than one member in the group, which in fact is generally the case when group attributes
are used, then the order of return would be random. Or if there had been binding information associated with both attributes, then here also the order in which binding handles would be returned would be random; that is, the caller might get a handle from the simple name attribute first, and then the handles exported to the group members, or it might get one or more of the group's member's handles, then one or more of the simple entry's handles, and so on.

6. **rpc_ns_binding_import_done()**

To delete the binding import context.

Figure 132 illustrates this activity.

---

**Figure 132. Importing from a Model That Uses Grouped Bindings**

The client shown in the figure imports a binding for object “A.” This becomes (through the group attribute) a referral back to the server’s entry where the bindings are held, and a binding is indirectly imported from the server entry. The object UUID for “A” is read, in a separate operation, directly from the object’s entry. With this information in its binding handle, the client makes its first remote call through the server’s interface. The call finds its way to the endpoint mapper via the partial binding information, and the endpoint mapper completes the binding by looking up the object UUID, which was registered there by the server.

**Global Organization of the Namespace**

Since DCE is designed to support very large namespaces, it uses a hierarchical service for binding. The global scale is separated into cells whose boundaries are administratively defined. For example, a company using DCE might have a cell containing its employees and local services. The cell namespace administrator could decide to put all the service entries in a single directory if the cell were small.

Both the import and export name service operations support default values derived from environment variables; for example, `RPC_DEFAULT_ENTRY_NAME`. The environment variables can be set by start-up files to the name of a well-known directory within the cell. The only remaining decision then will be how to name the actual entries within the directory. One easy method is to use mnemonic names, or names of interfaces such as `binop`, `spm_library`, and so on. If these entries are only being accessed by clients through profiles, their names will not be directly visible to the client anyway.

But now imagine a larger organization. The administrator will want to define some naming hierarchy based on geography, organization, or other criteria. Somewhere within this hierarchy some writable
directories (or parent directories) would be created, which could contain server entries, profiles, and so on. If clients are using only profiles to access bindings, then this organization will still be transparent to them. If clients want to bind to specific servers or objects, then more attention must be paid to the names given the servers’ or objects’ entries. The names should in some way reflect the organization, geography, or other relevant aspects of the server or object.

In summary, the important points to keep in mind are the following:

- The model should be appropriate for the organization and permit efficient administration of the namespace.
- There should be simple guidelines for naming objects and services so that users have a good chance of guessing the right answer.
Chapter 8. RPC Parameters

The RPC mechanism attempts to provide a data model as close as possible to the familiar local call model. For example, you can pass data “by reference” — by passing a pointer to a data item — despite the fact that client and server do not share an address space. Nevertheless, there are significant differences in both the syntax and semantics of RPC parameter data compared with C language local call data. For example, RPC provides directional attributes, conformant arrays, discriminated unions, and pipes, constructs which have no equivalents in C. Each requires an IDL specific syntax and has new semantics. Also, familiar constructs, such as pointers, closely mimic their local C language counterparts, but nevertheless must behave differently in some circumstances.

The DCE RPC programmer is thus confronted with a number of unfamiliar style and policy issues. The policy issues have mainly to do with which data types to use in given circumstances: for example, would you be better off using an array or a pipe to transfer a large block of data? This chapter contains recommendations that should help you make such choices. The style issues arise from the rich and unfamiliar syntax for RPC parameters which can make the mechanics of using many of the RPC data types seem rather daunting. This chapter contains numerous examples of basic data passing styles.

Execution Semantics

Before we begin to discuss the RPC data types themselves, a slight digression is necessary. Whatever data you pass, all RPCs must deal with the unreliable nature of remote network connections. A call may not complete due to a network failure, possibly leaving the call operations in an indeterminate state. For this reason, the IDL provides execution semantics attributes that applications can use to request certain (limited) guarantees about call completeness.

Ideally, in order for an application to behave in a determinant fashion, each operation needs to be invoked exactly once each time it is invoked. This requirement can be relaxed somewhat for idempotent operations: those which have the same effect when they are invoked one or more times. In this case, an application can settle for at-least-once semantics.

Unfortunately, with a remote procedure call, there is no way to guarantee either exactly once or at-least-once call semantics. Instead, RPC provides at-most-once and idempotent semantics. When a call completes and returns to the client, then at-most-once semantics is equivalent to exactly-once semantics, and idempotent semantics is equivalent to at-least-once. When a call fails to return to the client—either because of a server or communications failure—then the semantics make the following guarantees:

- **at-most-once**: The call was invoked on the server either 0 or 1 times. If the call was invoked, it may or may not have completed execution.
- **idempotent**: The call was invoked on the server 0 or more times. If the call was invoked, it may or may not have completed execution for any invocation.

In reality, idempotent semantics provides no guarantee for calls that fail to return to the client. In fact, DCE provides no guarantee about how idempotent semantics are actually implemented. It is perfectly correct to implement idempotency by using at-most-once semantics, and depending on protocol and implementation, this may be the case. Idempotent semantics is therefore really a hint from the application that a call is a candidate to be retried if the implementation uses a retry strategy.

These characteristics lead to two kinds of policy guidelines for call semantics. The first has to do with the behavior required of idempotent operations. An operation is a good candidate for idempotent semantics if it either changes no state on the server (for example, a read operation), or if the server state will be the
same even if the same call is invoked more than once (for example, a call that writes the same record) with the same [in] data. Note that in either of these cases, the result returned by a call may not be the same on each retry, since some other thread or process may have modified server state. A server that allows simultaneous reads and writes provides a good example. However, the runtime does guarantee commutativity of operations on the same association: an idempotent call will not be retried if a later call on the same association has been invoked.

The second policy issue has to do with how applications respond to call failures. The issues are the same for idempotent and at-most-once calls. In neither case can the client know whether the server manager operation was invoked, and, if it was invoked, whether it was completed. This leads to three possible failure states:

1. The manager operation was not invoked.
2. The manager operation was invoked but did not complete.
3. The manager operation was invoked and completed, but failed to return to the client.

The burden of determining which state applies, and implementing recovery actions rests almost entirely with the application. The RPC mechanism provides limited support for cleanup in the case of applications that use context handles to maintain state between calls. Application provided context rundown routines will be called on behalf of the application if a communications failure is detected. Beyond this rather elementary mechanism, DCE RPC does not provide any internal support for transaction processing, roll-back, or other recovery mechanisms. For applications where error recovery and maintenance of consistent state is essential, these must be implemented by the application programmer. The topic is beyond the realm of this policy guide.

IDL also provides two execution semantic attributes of somewhat more limited use: broadcast and maybe. Broadcast semantics may be used with connectionless transports when there are multiple servers on the local network that can handle a call. The client broadcasts the call request to all servers, and completes the call with one of them. Maybe semantics provides a calling style that may be used when a call has no [out] or [in, out] parameters. The call is attempted once, and no response is returned. Both broadcast and maybe semantics implicitly require that the operation be idempotent.

Parameter Semantics

RPC calls and the RPC API specify directional attributes for their parameters, even though such attributes are not formally supported by C. As a general rule, an [in] parameter is one that must be passed with a meaningful value and an [out] parameter is one whose value will be changed by the call. An [in,out] parameter is therefore one which must have a meaningful value on input and which may be changed on output.

The following table summarizes parameter semantics:

<table>
<thead>
<tr>
<th>Semantics</th>
<th>Meaningful Value Input</th>
<th>Changed on Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>[in]</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>[out]</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>[in,out]</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

An [out] or [in,out] parameter is one whose value is changed by the call, so it must be passed by reference, that is, as a pointer to the datum of interest. RPCs and the RPC APIs therefore always specify
output parameters as pointers. The address passed must always point to valid storage. For example, the ubiquitous status parameter may be declared in the IDL as

```plaintext
[out] error_status_t *status
```

The application code then needs to declare a variable such as

```plaintext
error_status_t st;
```

and pass it as &st to each RPC.

When a call allocates storage for an output parameter, it is declared as a pointer to a pointer. For example,

```plaintext
crpc_binding_vector_t **binding_vector
```

The application follows the same rule as in the status case, declaring a variable such as

```plaintext
crpc_binding_vector_t *binding_vec
```

and then passing this as &binding_vec. This obeys all the rules for output parameters: the address passed to the call points to valid storage, but the contents of that storage need not contain a meaningful value (in this case, need not be a valid pointer). A simple rule of thumb for output parameters is to declare a variable with one less asterisk than contained in the IDL (or RPC API) declaration and pass its address when calling the operation.

**Parameter Memory Management**

RPC attempts to extend local procedure call parameter memory management semantics to a situation in which the calling and called procedure no longer share the same memory space. In effect, parameter memory has to be allocated twice, once on the client side, once on the server side. Stubs do as much of the extra allocation work as possible so that the complexities of parameter allocation are transparent to applications. In some cases, however, applications may have to manage parameter memory in a way that differs from the usual local procedure call semantics.

For the purposes of memory allocation, three classes of parameters must to be considered:

- Nonpointer types
- Reference pointers
- Full pointers

For all types, the client application supplies parameters to the client stub, which marshalls them for transmission to the server. The client application is entirely responsible for managing the memory occupied by the passed parameters. On the server side, the server stub allocates and frees all memory required for the received parameters themselves.

In the case of the pointer types, however, the application and stubs must manage memory not only for the parameters themselves, but also for the pointed-to nodes. In this case, the memory management requirements depend both on the pointer type and on the parameter's directional attributes.

The rules are as follows:
Client Side Allocation

**in** parameters
For all pointer types, the client application must allocate memory for the pointed-to nodes.

**out** parameters
For reference pointers, the client application must allocate memory for the pointed-to nodes, unless the pointer is part of a data structure created by server manager code. For parameters containing full pointers, the stub allocates memory for the pointed-to nodes.

**in,out** parameters
For reference pointers, the client application must allocate memory for the pointed-to nodes. For full pointers, on making the call, the client application must allocate memory for the pointed-to node. On return, the stub keeps track of whether each parameter is the original full pointer passed by the client, or a new pointer allocated by the server. If a pointer is unchanged, the returned data overwrites the existing pointed-to node. If a pointer is new, the stub allocates memory for the pointed-to node. When a parameter contains pointers, such as an element in a linked list, the stub keeps track of the chain of references, allocating nodes as necessary.

It is the client application's responsibility to free any memory allocated by the stub for new nodes. Clients can call the routine `rpc_sm_client_free()` for this purpose.

If the server deletes or eliminates a reference to a pointed to node, an “orphaned” node may be created on the client side. It is the client application's responsibility to keep track of memory that it has allocated for pointed-to nodes and to deal with any nodes for which the server no longer has references.

Server Side Allocation

**in** parameters
For all pointer types, the stub manages all memory for pointed-to nodes.

**out** parameters
For reference pointers, the stub allocates memory for the pointed-to nodes as long as the size of the targets can be determined at compile time. When the manager routine is entered, such reference pointers point to valid storage. For parameters that contain full pointers, the server manager code must allocate memory for pointed-to nodes. Servers can call the routine `rpc_sm_allocate()` for this purpose.

**in,out** parameters
For reference pointers, the stub allocates memory for pointed-to nodes if either the size of the pointed to nodes can be determined at compile time or the reference pointers point to values received from the client. When the manager routine is entered, such reference pointers point to valid storage. For full pointers, the stub allocates memory for the original pointed-to nodes. The server manager code must allocate memory if it creates new references. Servers can call the routine `rpc_sm_allocate()` for this purpose.

The server stub automatically frees all memory allocated with `rpc_sm_allocate()`.

RPC Data Types

IDL provides both a number of primitive data types — such as various sizes of integers and floats, bytes, and Booleans — as well as pointers and a variety of constructed types based on the primitive types. The use of the primitive types is quite straightforward. The only important policy issues have to do with IDL data type to C data type mappings and with character handling. Pointers and the constructed types raise many more policy and style issues, and the bulk of this chapter is devoted to describing them.
IDL to C Type Mappings

Many of the primitive C data types represent items of different sizes on different machines. For example, an int may be 16 bits on one machine and 32 bits on another. These ambiguities can cause portability problems for some C programs, and they are intolerable for RPC programs. A parameter to an RPC call must represent the same size data item on both the client and server machine, whatever the machine architectures.

This means that when IDL declarations are compiled to generate C language headers and stubs, a given IDL type must always be declared in the corresponding C code as a C type of a specific length, no matter what machine the IDL compilation is done on. To achieve this:

1. Each IDL primitive type is always represented in the generated C files, by a specific defined C type
2. Each of the specific defined C types is defined by the local implementation of DCE so that it represents a data type of the correct length.

For example, a parameter declared in the IDL as a short, will be declared in the IDL generated header file as the defined type idl_short_int. Each implementation of DCE then defines the idl_short_int type correctly for the local C compiler and machine architecture to be an integer 16 bits long. For example, on a 32-bit machine, the idl_short_int type is typically defined as a short int.

When you write application code that refers to a parameter declared in the IDL, you must use a type that declares a data item of the same length. The safest policy is to use the same specific defined C type used in the headers and stubs. For example, if your IDL file declares:

```c
void my_op([in,out] short var);
```

Then your server manager code would contain a function that looks something like this:

```c
void my_op(idl_short_int var)
{
    ...
    ...
}
```

On a 32-bit machine, your code could probably use a short safely (because that is how your implementation probably defines idl_short_int, but such usage is not portable to other machine types and is therefore not recommended).

For numeric data, outgoing data is sent as is on the wire on the DCE network, and it is the receiver who converts the incoming data to the local representation (receiver-makes-right). This conversion is done automatically by DCE RPC using the Network Data Representation Protocol (NDR), which defines how the structured values supplied in an RPC call are encoded into byte stream format for transmission over the wire.

The following table shows the IDL to NDR to C type mappings for the IDL primitive types.

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>NDR Type</th>
<th>Defined C Type</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>idl_boolean</td>
<td>unsigned char</td>
</tr>
<tr>
<td>char</td>
<td>character</td>
<td>idl_char</td>
<td>unsigned char</td>
</tr>
<tr>
<td>byte</td>
<td>uninterpreted octet</td>
<td>idl_byte</td>
<td>unsigned char</td>
</tr>
<tr>
<td>small</td>
<td>small</td>
<td>idl_small_int</td>
<td>char</td>
</tr>
<tr>
<td>short</td>
<td>short</td>
<td>idl_short_int</td>
<td>short int</td>
</tr>
<tr>
<td>IDL Type</td>
<td>NDR Type</td>
<td>Defined C Type</td>
<td>C Type</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>----------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>long</td>
<td>long</td>
<td>idl_long_int</td>
<td>long int</td>
</tr>
<tr>
<td>hyper</td>
<td>hyper</td>
<td>idl_hyper_int</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>16- or 32- Bit Machines: Big Endian: struct { long high; unsigned long low; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Little Endian: struct { unsigned long low; long high; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64-Bit Machines: long</td>
</tr>
<tr>
<td>unsigned small</td>
<td>unsigned small</td>
<td>idl_usmall_int</td>
<td>unsigned char</td>
</tr>
<tr>
<td>unsigned short</td>
<td>unsigned short</td>
<td>idl_ushort_int</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>unsigned long</td>
<td>unsigned long</td>
<td>idl_ulong_int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>unsigned hyper</td>
<td>unsigned hyper</td>
<td>idl_uhyper_int</td>
<td>16- or 32-Bit Machines: Big Endian: struct { unsigned long high; unsigned long low; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Little Endian: struct { unsigned long low; unsigned long high; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>64-Bit Machines: unsigned long</td>
</tr>
<tr>
<td>float</td>
<td>float1</td>
<td>idl_float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>idl_double</td>
<td>double</td>
</tr>
<tr>
<td>handle_t</td>
<td>not transmitted</td>
<td>handle_t</td>
<td>void *</td>
</tr>
<tr>
<td>error_status_t</td>
<td>unsigned long</td>
<td>idl_ulong_int</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>ISO_LATIN_1</td>
<td>uninterpreted octet</td>
<td>ISO_LATIN_1</td>
<td>byte</td>
</tr>
<tr>
<td>ISO_MULTI_LINGUAL</td>
<td>(Note 1.)</td>
<td>ISO_MULTI_LINGUAL</td>
<td>struct{ byte row; byte column; }</td>
</tr>
</tbody>
</table>
In addition to the IDL primitive type mappings defined in the table, implementations provide a set of convenient typedefs that map the listed defined types into types that explicitly name amounts of storage. These are defined in IDL as:

- `typedef unsigned small unsigned8;` /* positive 8-bit integer */
- `typedef unsigned short unsigned16;` /* positive 16-bit integer */
- `typedef unsigned long unsigned32;` /* positive 32-bit integer */
- `typedef small signed8;` /* signed 8-bit integer */
- `typedef short signed16;` /* signed 16-bit integer */
- `typedef long signed32;` /* signed 32-bit integer */
- `typedef unsigned32 boolean32;` /* a 32-bit boolean */

They are defined in C as:

- `typedef idl_usmall_int unsigned8;`
- `typedef idl_ushort_int unsigned16;`
- `typedef idl_ulong_int unsigned32;`
- `typedef idl_small_int signed8;`
- `typedef idl_short_int signed16;`
- `typedef idl_long_int signed32;`
- `typedef unsigned32 boolean32;`

As a matter of programming style, these types have the advantage that the size of the declared data items is explicitly stated. For this reason their use in both IDL declarations and application C code is recommended. Note also that the IDL typedef

```c
typedef unsigned long error_status_t;
```

and the C typedef

```c
typedef idl_ulong_int error_status_t;
```

are also made available by implementations a convenient portable declaration for status parameters.

---

**Table 10 (Page 3 of 3). IDL/NDR/C Type Mappings**

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>NDR Type</th>
<th>Defined C Type</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO_UCS</td>
<td>ISO_UCS</td>
<td>struct{</td>
<td></td>
</tr>
</tbody>
</table>
|            |            |     byte group;
|            |            |     byte plane;
|            |            |     byte row;
|            |            |     byte column; } |

**Note:**

1. **Floating Point Data:** z/OS DCE always converts floating point data received from the network to IBM long and short format for the local z/OS system.
2. **Integer Data:**

   NDR represents signed integers in two's complement notation, and unsigned integers as unsigned binary numbers. The byte sequence of integer data is represented in two ways:

   - **big-endian format** Consecutive bytes of the byte stream representation are ordered from the most significant byte to the least significant byte.
   - **little-endian format** Consecutive bytes of the byte stream representation are ordered from the least significant byte to the most significant byte.

   If the ordering of bytes is different between machines, DCE RPC automatically converts the data at the receiver's end to the local system representation.

In addition to the IDL primitive type mappings defined in the table, implementations provide a set of convenient typedefs that map the listed defined types into types that explicitly name amounts of storage. These are defined in IDL as:

```c
typedef unsigned small unsigned8; /* positive 8-bit integer */
typedef unsigned short unsigned16; /* positive 16-bit integer */
typedef unsigned long unsigned32; /* positive 32-bit integer */
typedef small signed8; /* signed 8-bit integer */
typedef short signed16; /* signed 16-bit integer */
typedef long signed32; /* signed 32-bit integer */
typedef unsigned32 boolean32; /* a 32-bit boolean */
```
Character Data Handling

When passed as an RPC parameter, the IDL `char` type is automatically subject to ASCII-EBCDIC conversion, depending on the character encodings used by the client and server machines. Therefore, the contents of a `char` type may not be the same for the sender and receiver. This allows clients and servers to maintain the same semantics when passing characters between machines that use different encodings. For example, the character ‘a’ is represented by a byte with the value 61h on an ASCII machine, and a byte with the value 81h on an EBCDIC machine. RPC automatically makes the conversion so that a character parameter that prints as ‘a’ on the client machine also prints as ‘a’ when received by the server.

However, if what your application really intends is to pass a byte with the value 61h from client to server, such translation is clearly not what you want. When exchanging character data on-the-wire, you must ensure that any data you declare as `char` is true character data and does not imbed complex attributes, binary data or structures. To avoid any potential problem, when passing byte data with noncharacter semantics, use the IDL `byte` type.

In addition, ensure that if you explicitly use the `unsigned` keyword when specifying the `char` data type in an IDL definition, you must also use it within the client, server, and manager code.

Also note that IDL provides three international character types for use with non-ASCII, non-EBCDIC character sets: `ISO_LATIN_1`, `ISO_MULTI_LINGUAL`, and `ISO_UCS`. To ensure portability, your application should use these types to declare character data in one of these sets.

**DBCS Character Data:** Your DCE applications may need to exchange Double-Byte Character Set (DBCS) data between the client and server. For example, your application may need to exchange data using an international character set such as Kanji. To handle double-byte character data, such as Kanji, use the `byte` attribute in the IDL file to declare the data type. This will prevent any data format conversion by the DCE RPC mechanism.

**Pointers**

RPC pointers differ from local pointers in one key respect: there is no shared address space between client and server. This means that the stubs need to marshall the pointed-to data itself. To do so, the stubs must be able to dereference any pointer passed as a parameter. This means that a pointer, even if it does not point at useful data, must be initialized either to `NULL` or to a valid address before it is passed as a parameter. This behavior may be counter-intuitive for programmers used to local procedure calls, where pointers may be freely passed whether they have been initialized or not, and is a common source of programming grief for remote procedure calls.

To be able to marshall pointer referents, the stubs need to know, either at compile time or at run time, how much data to transmit; that is, they need to know the size of the pointed to object. This can require a good deal of work on the part of the stubs in the case of varying or conformant arrays and objects like linked lists.

One effect of this is that pointers only reference the marshalled data itself; that is, data of the size determined by the stub. For example, passing an `idl_char *` parameter causes the stub to marshall a single `idl_char`, since that is the size of the object pointed to by an `idl_char *`. Typically, a local procedure call passes a `char *` type in order to pass the address of an array of characters, not a single `char`; but a remote routine that tries to move such a pointer beyond the transmitted `char` will very likely find itself pointing to invalid storage and certainly not to the intended string.

A similar case is illustrated in the sample code: a client passes an array and an `[in,out, ptr]` pointer to an array element. If the server sets the pointer to point to some element of the passed array, then it will point to memory holding a copy of that element when the call returns to the client. It will not point to any part of
the passed-in array itself, and any attempt to increment or decrement the pointer on the client side will
leave it pointing to an invalid location.

This is one example of the fact that you cannot assume that the results of pointer arithmetic will be the
same for a local and remote procedure call. To give another example, suppose a call passes two
parameters: a data structure and a pointer to the type of the data structure, set to \texttt{NULL}. If the server
application then sets the pointer to point to the data structure, the client stub will allocate new storage for
the returned data structure and set the returned pointer to point to it. As a result, the returned pointer will
not point to the original structure, but to a copy of it in stub maintained memory.

This may seem like an IDL limitation, but in fact, the real issue is that the client and server address
spaces are different, and some operations in one address space cannot be reflected in the other.
Specifically, the server application cannot meaningfully interpret an address in the client address space,
and vice versa. So, as in the last example, the server cannot set a pointer to point to a structure in the
client address space; it can only ask the client stub to mirror any changes made at the server.

**Memory Allocation Routines:** The stubs will do their best to allocate any new memory required
for marshalled pointed-to nodes so that the marshalling is transparent to the application. On the server
side, stub allocated memory exists for the scope of the manager routine call. The stub frees such memory
once the nodes have been marshalled. On the client side, however, the stubs obviously cannot free the
memory they have marshalled since they are returning the data to the client application. Therefore, in
order to avoid memory leaks, when a client makes an RPC that results in the client stub allocating
memory, the client application needs to call \texttt{rpc_sm_client_free()} to free the pointed to memory.

When a server manager routine needs to allocate new memory for a pointed-to node, it can do so either
statically or by making a call to a memory allocation routine. In the latter case, however, the manager
cannot deallocate the memory it has allocated, since the pointer must be valid when the call returns (so
that the stubs can marshall the data). Only the stubs can free such memory. In order to permit this, server
managers need to call \texttt{rpc_sm_allocate()} to allocate memory for parameters. The stubs free all memory
allocated by \texttt{rpc_sm_allocate()} once they have marshall the required data, thus avoiding memory
leaks.

**Pointer Types:** For reasons of efficiency, IDL distinguishes between reference [\texttt{ref}], full [\texttt{ptr}], and
unique [\texttt{unique}] pointers. As we saw above, even though pointers are used by applications to pass data
by reference, the lack of shared address space means that the stubs have to pass the data by value and
provide the receiver with a reference to the passed data.

In the simplest case, a pointer always points to the same memory: that is, its value does not change. In
such a case the stubs always marshall the passed value from and to the same memory location on the
sender and receiver respectively. This style of marshalling is provided by [\texttt{ref}] pointers.

When the value of a pointer changes during a call, the stubs have a more complex task. Suppose, for
example, an [\texttt{in, out}] pointer is \texttt{NULL} before an RPC and is set by the server application to point to some
data structure allocated by the server. As in the [\texttt{ref}] pointer case, the server stub needs to marshall the
(new) referent and the client to unmarshall it, but the client stub also needs to do two more things: it
needs to allocate space for the unmarshalled referendum, and it needs to point the previously \texttt{NULL} pointer to
it. Similarly, for a pointer that initially points to one memory location and is changed during an RPC to
point to another, the client stub needs to allocate new memory to hold the unmarshalled value of the new
referent and to change the pointer value accordingly. Not all of the extra work is confined to the client
stub either. Obviously, the client stub needs to find out that the value of the pointer has changed, so the
server has to marshall, and the RPC protocol has to transmit, extra data to indicate this. This style of
marshalling is provided by full ([\texttt{ptr}]) pointers, and it obviously requires more overhead than reference
pointer marshalling.
Unique pointers provide for an intermediate case: a pointer that always points either to a single memory location or is NULL. Such a pointer may change from NULL to a non-null value or from a non-null value to NULL, but never has more than one non-null value. Such a pointer is marshalled more efficiently than a full pointer, but not as efficiently as a reference pointer.

Applications should consider the [ref] and [unique] pointer types as optimizations. A full [ptr] pointer can always be used. The [ref] and [unique] pointer types may be used whenever the application is guaranteed to meet the restrictive conditions under which these types work.

As a guide to using the pointer types, there are a few general rules and a number of special cases, having mainly to do with embedded pointers and data of variable size. The rules are:

- In passing parameters, you need to distinguish carefully between top-level and lower-level pointers. A top-level pointer is a pointer passed as an argument to a call. A lower-level pointer is one contained in the referent of a top-level pointer. The directional semantics [out] and [in, out] both require parameters to be passed by reference and hence always require a top-level pointer.

  The model is, essentially, that the client provides a container into which the returned value is written. In the [out] parameter case, the contents of the container are assumed to be unimportant on input and are not marshalled by the client stub. In the [in, out] case, the contents are assumed to be meaningful and are passed to the server.

  The top-level pointer is thus the address of the parameter “container,” and obviously, this value should not change during the course of the call. If it did, the return value would be written to some undetermined place in the client address space. Hence, the top level of [out] and [in, out] parameters have reference pointer semantics. The IDL compiler enforces this for [out] parameters by permitting only the [ref] attribute. It does not force this for [in, out] parameters, but the behavior is exactly the same. Remember, the actual parameters of an RPC call are always passed by value: hence a call cannot change the value of a top-level pointer. It can only change the value of something passed by reference.

- To pass an [in] parameter by reference, you can pass its address as a pointer of either style. The server stub will allocate and deallocate the required memory for the pointer referent. Since an [in] pointer has no reason to change its value, it is at least slightly more efficient to use a reference pointer in this case.

- Since [out] semantics do not consider the contents of such storage to be meaningful, an [out] parameter is not marshalled on the call. The server stub will allocate memory to hold the referent as long as the size of the referent is known at compile time. The stub obviously cannot allocate memory for referents whose size is determined arbitrarily by the server application. For such parameters (such as linked lists) the server application must allocate space. One tricky case to consider is a linked list. The server stub allocates space for the head element, since it knows the size of such an element. The server manager then allocates space for the remaining elements and marshalls them back to the client. The client stub will allocate all necessary space for the server-created receive parameters.

  A server-created structure may contain reference pointers which the server may then set to point to objects it also allocates. All of this will be mirrored by the client stub. Note that this does not violate the rules for reference pointers, since the contained pointers do not change value during the call; they are created by the server application and passed back to the client exactly the same way that top-level reference pointers are created by clients and passed to servers.

  When an application wishes to have the callee allocate space for an [out] parameter, it needs to use two levels of indirection: a reference pointer to a full pointer to the data structure to be allocated. The client allocates the full pointer, setting it to NULL, and passes its address to the call. The server application then allocates the data structure and sets the full pointer to point to it. The client stub will then allocate space for the data structure on the return.
• An [in, out] reference pointer behaves exactly like the [out] reference pointer case, except that the server stub may be able to allocate space for a referent even if its size is not known at compile time. This will be the case when the client application creates an instance of a variable sized referent, such as a linked list. In such a case, the server stub will allocate sufficient space for the referent supplied by the client.

• You must take care when a server deallocates a [ptr] pointer referent. For an [in, out] parameter, the client-side stub does not deallocate the client-side referent, but the application should treat the referent as undefined, as if, in effect, the deallocated pointer referent had been unmarshalled by the client stub. By default, in the case of an [in] parameter, the value of the pointer referent remains unchanged on the client side. However, this default behavior can be modified by applying the [reflect_deletions] attribute to the operation. In this case, the client-side stub will deallocate the pointer referent. The client and server must use the rpc_sm* routines to allocate and free memory for this reflection of deletions to work.

• For an [in, out] parameter which is a [ptr] pointer, if the server sets the parameter value to NULL, the client will no longer be able to dereference the pointer on return. If the client has no other means to reference the original pointed-to node, the node is said to be “orphaned”: the client will be unable to free it.

**Pointer Examples:** The following sample code demonstrates the basic properties of pointers. The first example demonstrates pointer arithmetic and how changes in the server address space can be reflected back to the client using full pointers. In the .idl file we declare a type that is an array of three integers, and a type that is a pointer to an integer. The operation takes the array as an [in] parameter and the pointer as an [in, out] parameter.

```c
const unsigned32 ARRAY_SIZE = 3;
typedef unsigned32 num_array[ARRAY_SIZE];
typedef [ptr] unsigned32 *num_ptr;

void ptr_test1(
    [in] handle_t handle,
    [in, out] num_ptr *client_ptr,
    [in ] num_array client_array,
    [out] error_status_t *status )
);
```

The server manager code to implement this points the client pointer to the beginning of the array and then increments it once:

```c
void ptr_test1(
    handle_t h,
    num_ptr *client_ptr,
    num_array client_array,
    error_status_t *status )
{
    *status = 0;
    *client_ptr = client_array;
    ++(*client_ptr);
}
```

On return, the client's version of the pointer will point to memory that holds the second element of the array. It will not point to the array itself, however. The client code demonstrates this:
num_ptr client_ptr = NULL;
num_array client_array = {25, 50, 75};

ptr_test1(binding_h, &client_ptr, client_array, &status);

/*
 * The test function pointed the client pointer to the
 * second array element. On return, this points to memory
 * that holds this value.
 */

printf("Client pointer points to %i\n", *client_ptr);

/* However, if we now increment the pointer, it
 * points to uninitialized memory. This shows the
 * limits of mirroring.
 * *** WARNING: You may dump core here !! ***
 */

client_ptr++;
printf("Client pointer now points to %i\n", *client_ptr);

What happens here is that the client stub allocates space for the new referent of client_ptr when the call returns. This space now holds the value in the second element of the array. The pointer no longer points to the original array but to this newly allocated space. You can see this clearly when the client attempts to increment the pointer. Instead of pointing to the third element of the array, it points to some undetermined place in memory, and the client may fail when it tries to dereference the pointer.

As an exercise, you could change the code to declare a pointer to the num_array defined type rather than to an integer. Then you could have the server manager point this to the input array and return it without incrementing the pointer. The returned pointer will now reference a copy of the original client array with all its elements. It will not, however reference the original array itself.

The second pointer example illustrates passing a linked list. The .idl declaration is as follows:

typedef struct link {
    unsigned32 value;
    [ptr] struct link *next;
} link_t;

void ptr_test2(
    [in] handle_t handle,
    [in, out, ref] link_t *head,
    [out] error_status_t *status
);

The server manager code is as follows:

void
ptr_test2(
    handle_t handle,
    link_t *head,
    error_status_t *status
)
{
    link_t *element;

    if (head)
    {
        element = head;
        while (element->next)
            element = element->next;

        /* Code to use the linked list goes here */
    }

    /* Code to return the pointer goes here */
}
The manager operation adds a new element to the end of the linked list. Note that the head parameter has [in, out] semantics here: we must pass in a pointer to a valid element. (The next example shows how to implement an [out] parameter that is allocated by the operation.)

In this and the following example, we use \texttt{rpc\_sm\_allocate()} to allocate data on the server side. This gives the semantics you probably want for a dynamically allocated referent for a pointer parameter: on return, the data is automatically deallocated on the server, and further manager operations that access this data do so via a pointer parameter passed by the client. Memory leaks on the server are thus avoided.

An application must be very cautious if it attempts to use pointer parameters in a way that contradicts such semantics: for example, by returning a pointer to static global storage on the server. In such a case, the server and client versions of such storage can easily become inconsistent. A context handle, which the client must not modify, is typically what you want in such a case.

The client code for the linked list test is:

```c
link_t first, *element;
int i;
first.value = 2;
first.next = NULL;
for (i = 0; i < 8; i++)
  ptr_test2(binding_h, &first, &status);

element = &first;
while (element->next)
  {
    printf("%i, ", element->value);
    element = element->next;
  }
printf("%i\n", element->value);
```

The client passes in the head element, and then calls the server several times to add more elements to the list. Finally, the client prints out the list.

The next pointer example illustrates how the stubs automatically allocate memory for an [out] parameter. The client application allocates a \texttt{NULL} pointer to the data structure of interest and passes the address of this pointer as the [out] parameter. The server manager allocates a structure, and on return the client stub allocates it too, automatically.

The .idl declaration is as follows:

```c
typedef struct {
  [ref] unsigned32 value;
} number;
typedef [ptr] number *number_ptr;
```
void ptr_test3(
    [in] handle_t handle,
    [out, ref] number_ptr *client_ptr,
    [out] error_status_t *status
);

The server manager operation is then:

```c
void
ptr_test3(
    handle_t handle,
    number_ptr *client_ptr,
    error_status_t *status
){
    number_ptr nptr;
    unsigned32 *nval;
    nptr = (number_ptr) rpc_sm_allocate(sizeof(number), status);
    if (*status == rpc_s_ok)
    {
        nval = (unsigned32 *) rpc_sm_allocate(sizeof(unsigned32), status);
        if (*status == rpc_s_ok)
        {
            *nval = 256;
            nptr->value = nval;
            *client_ptr = nptr;
        }  
        *status = error_status_ok;
    }
if (*status == rpc_s_ok)
};
```

The client test code looks like this:

```c
number_ptr client_ptr = NULL;
ptr_test3(binding_h, &client_ptr, &status);
printf("Value = %i\n", (unsigned32)* (client_ptr->value));
```

Note the use of [ref] pointers here. The top-level [ref] pointer (the one passed as a parameter to the call) must point to valid storage when the call is made even though the pointer is not marshalled when the call is made. This follows the rules for [ref] pointers: they may not be NULL and may not change value during a call. The returned structure also contains a [ref] pointer, and the client stub does automatically allocate space for its referent when the call returns. This is an exception to the rule that an [out] [ref] pointer must point to valid storage when the call is made. In this case, the pointer is embedded in a structure which is created by the server. As long as the top-level pointer points to valid storage (to hold the returned structure), the client stub will allocate space for the referents of any newly-created [ref] pointers that it contains.

The final example illustrates node deletion. The .idl declaration is as follows:

```
[reflect_deletions] void ptr_test4(
    [in] handle_t handle,
    [in, out, ptr] unsigned32 *number,
    [out] error_status_t *status);
```

The server code to implement this operation frees the memory pointed to by the input pointer and returns the pointer:

```c
void
ptr_test4(
    handle_t h,
```
unsigned32 g number, 
error_status_t *status 
) 
{ 
*number = 32; 
rpc_sm_free(number, status); 
*status = error_status_ok; 
};

The client code is as follows:
unsigned32 *num; 

rpc_sm_enable_allocate(&status); 
num = (unsigned32*) rpc_sm_allocate(sizeof(unsigned32), &status); 
if (*status == rpc_s_ok) 
{ 
*num = 64; 
ptr_test4(binding_h, num, &status); 
}

There are so many ways to use (and misuse) IDL pointers that it would be impossible to give a complete set of examples. The section on arrays contains more pointer examples.

Context Handles

Context handle semantics vary according to the application role. On the server side, the semantics are those of a full pointer. To the client application, a context handle has similar semantics to a fully bound server binding handle, except that the client may not perform any operations to modify it. To the client it represents a binding to context maintained by a specific a server instance. Because the context handle may also specify an object UUID, it may also bind to a specific type manager in the server instance; that is, a context handle refers to context maintained by a specific type manager in a specific server instance. It is valid over a series of calls within this scope. To enforce this, a context handle is intended to be passed as an explicit binding parameter for each operation that refers to the maintained context. Any attempt to use a context handle outside this scope will fail. Context handles are described in more detail as an RPC application development topic in the z/OS DCE Application Development Guide: Core Components.

Arrays

Array parameters provide an efficient way to pass contiguous blocks of data with little application overhead. The stubs take care of serializing and reassembling the passed data transparently to the application. When an application is interested in passing an entire buffer or some contiguous portion of a buffer synchronously — so that all of the data is made available to the receiver at the same time — arrays provide the most efficient mechanism. Pipes provide no advantage unless the data is to be processed asynchronously.

Arrays may be passed as RPC parameters, but, as in the case of other RPC data, the stubs need to know the size of data to be marshalled. The simple solution is to declare arrays of fixed size in the IDL. This can be inefficient however, since array sizes may vary at runtime, and since not all data in an array may need to be passed on every call. Therefore, IDL provides a variety of field attributes (max_is, min_is, size_is, last_is, first_is, and length_is) to permit the size and bounds of the marshalled data to be determined at runtime. Note that passing a pointer to an array is not any more efficient as a way to deal with the problem of varying array sizes. Remember that marshalling a pointer requires marshalling the pointer's referent, so the array data will be marshalled anyway. Note also that the IDL language does not permit declaring a pointer to a varying array.
The size of the array data marshalled is determined in one of two ways. In a conformant array, the size of the array is not declared in the IDL declaration, and one of the max_is or size_is attributes is used to determine the size of the marshalled data at run time. In a varying array, the size of the array is declared in the .idl file, but one or more of the other field attributes determines what range of elements is actually marshalled. Arrays may be both conformant and varying at the same time.

Each field attribute is associated with some variable whose value is known at run time. The scope of this association is within either an operation declaration or a structure declaration. That is, when the array is a parameter of an operation, the field attribute variables must also be parameters of the same operation. Similarly, when the array is a member of a structure, the field attribute variables must be members of the same structure.

The following samples show a series of array declarations using some of the many possible forms:

```c
/* An array of fixed size */

typedef char char5array[5];
typedef char5array *char5ptr;

void array_test1(
    [in] handle_t handle,
    [in] char5ptr a_pointer,
    [out] error_status_t *status);

/* A conformant array: the size is determined at run time */

void array_test2(
    [in] handle_t handle,
    [in] unsigned32 size,
    [in, size_is(size)] char an_array[],
    [out] error_status_t *status);

/*
 * A varying array: the portion of the array transmitted is
 * determined at run time
 */

typedef struct{
    unsigned32 first;
    unsigned32 length;
    [first_is(first), length_is(length)] char array[10];
}v_struct;

void array_test3(
    [in] handle_t handle,
    [in] v_struct v_array,
    [out] error_status_t *status);

/*
 * A conformant and varying array: both size and the portion
 * transmitted are determined at run time
 */

typedef struct{
    unsigned32 size;
    unsigned32 first;
    unsigned32 length;
}cv_struct;
```
The server manager sample code to test these declarations is as follows:

```c
void array_test1(
    handle_t handle,
    char5ptr a_pointer,
    error_status_t *status
)
{
    printf("Array test 1\n");
    printf("%c %c \n", (a_pointer)[gz'ro@ot], (a_pointer)[1]);
    *status = error_status_ok;
}

void array_test2(
    handle_t handle,
    unsigned32 size,
    idl_char an_array[],
    error_status_t *status
)
{
    unsigned32 i;

    printf("Array test 2\n");
    for ( i = gz'ro@ot; i < size; i++)
    {
        printf("%c ",an_array[i]);
        printf("\n");
    }
    *status = error_status_ok;
}

void array_test3(
    handle_t handle,
    v_struct v_array,
    error_status_t *status
)
{
    unsigned32 i;

    printf("Array test 3\n");
    for ( i = v_array.first; i < v_array.first + v_array.length; i++)
    {
        printf("subscript %i value %c\n", i, v_array.array[i]);
        *status = error_status_ok;
    }
}
```
void array_test4(
    handle_t handle,
    cv_struct *cv_array,
    error_status_t *status
)
{
    unsigned32 i;

    printf("Array test 4\n");
    for (i = (*cv_array).first; i < (*cv_array).first + (*cv_array).length; i++)
        printf("subscript %i value %c\n", i, (*cv_array).array[i]);
    *status = error_status_ok;
}

The client sample code is as follows:

char5array fixed_array = {'a','b','c','d','e'};

v_struct varying_array = {3,4,{'a','b','c','d','e','f','g','h','i','j'}};

struct {
    unsigned32 size;
    unsigned32 first;
    unsigned32 length;
    char array[10];
}cv_array = {10, 4, 5, {'a','b','c','d','e','f','g','h','i','j'}};

array_test1(binding_h, &fixed_array, &status);
array_test2(binding_h, 5, fixed_array, &status);
array_test3(binding_h, varying_array, &status);
array_test4(binding_h, &cv_array, &status);

The server output will look like this:

Array test 1
a b
Array test 2
a
b
c
d
e
Array test 3
subscript 3 value d
subscript 4 value e
subscript 5 value f
subscript 6 value g
Array test 4
subscript 4 value e
subscript 5 value f
subscript 6 value g
subscript 7 value h
subscript 8 value i
Note that for the last test, the declared structure contains a conformant and varying array. The C language does not provide any intrinsic support for conformant arrays, and the actual IDL-generated header declaration for the type `cv_struct` looks as follows:

```c
typedef struct {
    unsigned32 size;
    unsigned32 first;
    unsigned32 length;
    idl_char array[1];
} cv_struct;
```

The declared structure contains an array only one element in length. When creating an instance of this type, the application must allocate a data structure of the correct size, either statically, as in the sample client code (the data item `cv_array`), or dynamically. The recipient of such a data structure (in this case, the server manager code), can then determine the actual size of the marshalled data by examining relevant field attribute variables (in this case, the structure's `size` member). Note also that IDL requires a structure containing a conformant array to be passed "by reference"; that is, as a pointer referent.

Conformant and varying arrays provide a way to pass blocks of contiguous data of varying sizes and ranges. However, there is no intrinsic mechanism for passing sparse arrays efficiently. Applications may, however, supply their own mechanisms for compressing and passing large, sparse arrays using the `[transmit_as]` mechanism.

There are a number of complications that can arise when using arrays of pointers. For example, an `[out]` or `[in,out]` conformant array of pointers, accompanied by an `[out]` or `[in,out]` field attribute variable, could potentially be of any size when returned to a caller. For `[ref]` pointers, which may not be `NULL`, the client must therefore ensure that all possible returned pointers in such an array actually point to valid storage. You can easily avoid such complications by sticking to the more straightforward array usages discussed here.

**Structures and Unions**

There are no important policy issues relating to structures and unions as RPC parameters. Pointers and arrays as members of structures and unions are sometimes treated differently from separately declared types. By embedding pointers and arrays in structures and unions, you can sometimes achieve behavior that cannot be obtained by passing them as separate parameters.

Structures and unions can be used wherever they would be used in a non-RPC application. IDL structures differ from C language structures in one important respect: they may contain conformant arrays, which are not supported by C. A structure that contains a conformant array is itself conformant; that is, the size of the structure may not be determined until runtime. Applications need to do some extra work to determine the size of, and allocate, conformant structures. When structures are used to create linked lists and trees, the stubs do considerable work to insure that server allocated data is reflected back to the client.

IDL union syntax is quite different from C syntax, since IDL unions must be `discriminated` so that stubs can determine which of the contained data types to marshall. As with conformant and varying arrays, which use a field attribute variable to determine array size and bounds at run time, IDL unions use a `discriminator` variable to determine which data type is marshalled.

IDL unions may be `encapsulated` or `nonencapsulated`. In an encapsulated union, the IDL compiler packages the union type and the discriminator in a structure. In a nonencapsulated union, the IDL `[switch_is]` attribute is used to identify a discriminator variable. In this case, as in the case of array field attribute variables, the application must declare the discriminator and the union together, either as members of a structure or as parameters of an operation.
When a union is passed as a parameter, the value of the discriminator must either match one of the constants declared in the switch construct, or the switch must contain a default case. Otherwise, a stub marshalling error will occur.

Following are several examples of IDL union syntax. They are accompanied by the resulting IDL generated C header file declarations, and show how applications must refer to the union constructs declared in the IDL. The first example shows a set of declarations for an encapsulated union. The union holds either of two structures, one containing UUIDs, the other unsigned integers.

```c
typedef struct two_uuid_s_t {
    uuid_t uuid1;
    uuid_t uuid2;
} two_uuid_t;

typedef struct two_uint_s_t {
    unsigned32 uint1;
    unsigned32 uint2;
} two_uint_t;

typedef enum {
    uuids,
    uints
} union_contents;

typedef union switch (union_contents type){
    case uuids:
        two_uint_t integers;
    case uints:
        two_uuid_t ids;
} test_union_t;
```

The resulting IDL generated C header declarations look like as follows:

```c
typedef struct two_uuid_s_t{
    uuid_t uuid1;
    uuid_t uuid2;
} two_uuid_t;

typedef struct two_uint_s_t{
    unsigned32 uint1;
    unsigned32 uint2;
} two_uint_t;

typedef enum{
    uuids,
    uints
}union_contents;

typedef union{
    union_contents type;
    union {
        /* case(s): 0 */
        two_uint_t integers;
        /* case(s): 1 */
        two_uuid_t ids;
    } tagged_union;
} test_union_t;
```
The IDL compiler packages the encapsulated union as a structure with the discriminant as the first member. To pass the union as an [in] or [in,out] parameter, the calling application must set the type field of this structure to either of the enumeration values integers or ids. To return the union as an [out] or [in,out] parameter, the callee must similarly be sure that the value of the type field is correctly set. To discover which data type was marshalled, the recipient can check the value of the type field.

The following is an example of nonencapsulated union usage. The .idl declaration is as follows:

```idl
typedef [switch_type(long)] union {
    [case (1,3)] float a_float;
    [case (2)] short b_short;
    [default]; /* An empty arm */
} n_e_union_t;
```

The C header declaration of the nonencapsulated union generated by the IDL compiler is as follows:

```c
typedef union {
    float a_float;
    short b_short;
} n_e_union_t;
```

In this case, the discriminant must be separately declared in order for the union to be marshalled. The IDL [switch_is] attribute identifies the discriminant for an instance of the declared union type. Two examples of such .idl declarations follow:

```idl
/*
 * A structure that includes the union declared above and a member
 * that is used as the discriminant. This structure can be passed
 * as an RPC parameter.
 */

typedef struct {
    long a;
    [switch_is(a)] n_e_union_t b;
} a_struct;

/*
 * An operation declaration that passes the declared union type along
 * with a discriminant.
 */

void op1 (
    [in] handle_t h,
    [in, switch_is (s)] n_e_union_t u,
    [in] long s
);
```

Pipes

Pipes allow application-level optimization of bulk data transfer, by allowing the communication and processing of data to overlap. The actual data communications occur at about the same speed as arrays. However, pipes can reduce latency (how soon the application sees each “chunk” of data) and increase memory utilization. The intent is that the pipe routines should actually process the data and then get rid of it (for example, summarize it; write it to a file; pass it to another thread) rather than merely write it into an array. If an application desires to pass all of a stream of data and process it synchronously, then an array will probably be more efficient, since it entails considerably less processing overhead, as well as being
simpler to program. For more on pipes as a topic in RPC application development, see the
Application Development Guide: Core Components.

The transmit_as Attribute

The [transmit_as] attribute provides applications a way to do their own marshalling of data types. This is
primarily useful as a way to deal with data structures that the stubs cannot marshall efficiently, such as
sparse arrays. Following is an example of code to compress and reconstruct a large array by removing
and then replacing all the zero-valued elements:

The .idl declarations are as follows:

```c
/*
 * Transmit_as example: Here we turn a large sparse array into
 * a small conformant array for transmission. The server is able
 * to reconstitute the sparse array.
 */

const long int S_ARRAY_SIZE = 32;

typedef struct{
    unsigned32 value;
    unsigned32 subscript;
} a_element;

typedef struct{
    unsigned32 size;
    [size_is(size)] a_element array[];
} compact_array_t;

typedef [transmit_as(compact_array_t)] unsigned32 sparse_array_t[S_ARRAY_SIZE];

void ship_array(
    [in] handle_t handle,
    [in] sparse_array_t *array,
    [out] error_status_t *status
);
```

All the callback routines are placed in a single module that is linked with both client and server (in this
case, for the test interface). As an alternative, the appropriate callbacks could be declared separately
within the client and server modules:

```c
/*
 * test_xmit.c:
 * The routines required to implement a [transmit_as] type.
 */

#include "test.h"

/* The to_xmit routine must allocate all space for the transmitted
 * type. In general, the stubs have no way to determine how to allocate
 * space for the transmitted type. Here, for example, the to_xmit
 * routine determines the size of a conformant array.
 */

void sparse_array_t_to_xmit(sparse_array_t *s_array,
compact_array_t **c_array
{
    unsigned32 i, j;
    unsigned32 csize;

    /* Count up the number of non-zero elements in the sparse array */
    for (i = 0, csize = 0; i < S_ARRAY_SIZE; i++)
    {
        if ((*s_array)[i] != 0)
        {
            csize++;
        }
    }

    /* Allocate a structure to hold the compact array */
    *c_array = (compact_array_t *)calloc(csize*2 + 1, sizeof(unsigned32));
    ((compact_array_t **)c_array).size = csize;

    /* Fill in the compact array from the non-zero elements */
    for (i = 0, j = 0; i < S_ARRAY_SIZE; i++)
    {
        if ((*s_array)[i] != 0)
        {
            ((compact_array_t **)c_array).array[j].value = (*s_array)[i];
            ((compact_array_t **)c_array).array[j++].subscript = i;
        }
    }
}

/* The from_xmit routine may not have to allocate any space for the
* presented type. The presented type is always of a definite size
* (conformant, varying, etc. types are not permitted), so the stub
* provides an instance of the top level of the presented type. In
* this case, for example, s_array points to an instance of a sparse
* array. If the presented type contains any pointers, the from_xmit
* routine needs to allocate space for the referents and the free_inst
* routine needs to free them.
*/

void sparse_array_t_from_xmit(compact_array_t *c_array,
    sparse_array_t *s_array)
{
    unsigned32 i, j;
    for (i = 0; i < ((compact_array_t *)c_array).size; i++)
    {
        j = ((compact_array_t *)c_array).array[i].subscript;
        (*s_array)[j] = ((compact_array_t *)c_array).array[i].value;
    }
}

/* This routine is called to free anything allocated by the
* to_xmit routine.
*/
void sparse_array_t_free_xmit(compact_array_t *c_array) {
    free(c_array);
}

/* This routine is called to free anything allocated by the
 * from_xmit routine. Since from_xmit doesn't allocate anything
 * this is a null routine.
 */
void sparse_array_t_free_inst(sparse_array_t *s_array) {
}

The client code to exercise the sparse array transmitted type is as follows:

sparse_array_t test_array;

/* Create a sparse array with only three non-zero members */
memset(test_array, 0, sizeof(unsigned32) * S_ARRAY_SIZE);
test_array[0] = 2;
test_array[20] = 4;
test_array[31] = 8;

/*
 * When compressed, this array requires 7 32-bit integers, as opposed
 * 32 32-bit integers for the uncompressed array. If you don't care
 * about reconstructing the sparse array on the server side, you can
 * get even more efficiency.
 */
ship_array(binding_h, &test_array, &status);

The server manager code is as follows:

void ship_array(
    handle_t binding_h,
    sparse_array_t *array,
    error_status_t *status
) {
    int i;

    /*
    * Print the elements of the sparse array.
    */
    for (i = 0; i < S_ARRAY_SIZE; i++)
    {
        printf("%i\n", (*array)[i]);
    }
    *status = error_status_ok;
}

Note that the free_inst routine will not be needed if the transmitted type does not contain pointers.
However, the routine is called by the stub automatically in any case, so at least a null routine must be provided. As an exercise, you might add printf()s to each callback to see when it is called. You could
also add code to show the format of the transmitted array before it is reconstructed by the `from_xmit` routine. Finally, you can create an even more efficient compression by not attempting to reconstruct the original array on the server side.
Chapter 9. Server Management

Every DCE server requires some management. At a minimum, servers need to be started and stopped. In addition, servers usually provide generic server information such as the server principal name and an indication that the server is listening for remote calls. Servers may also permit other kinds of management operations while they are running; it is perfectly feasible to have a server reinitialize or even unregister and reregister endpoints while it is running.

From the management perspective, servers are thought of as either on-demand or persistent. In the on-demand model, a server only starts (thus occupying system resources) when it is needed. When an on-demand server is installed, a startup configuration is also installed with dced. Such a server would then use the configuration (obtained by a call to the dce_server_inq_server() routine) when it is auto-started by dced on receipt of an RPC request for an interface, operation, or object registered for that server.

A persistent server is one that runs continuously. Starting, stopping and otherwise managing such a server are typically considered privileged operations. In general, a robust persistent server should provide a separate application control program that calls the DCE management interfaces (APIs for dced, RPC, and the like) and the application's own management interface (if one is provided). Of course, a server cannot start itself, but an application control client program can start the server via the dced. The model looks similar to what is shown in Figure 133.

![Diagram](image-url)

Figure 133. Managing a Server with a Control Client

In addition to starting and stopping the server, dced's management routines provide other control operations. For example, the control program can use dced_server_disable_if() and dced_server_enable_if() to disable and re-enable specific interfaces offered by the server.

Application-specific management operations can be used to exert even finer control than is possible with the DCE-provided services.
Application Support for Server Management

Applications can support server management at three levels. At a minimum, every server automatically supports the RPC Management API (routines the begin with `rpc_mgmt_`). By attaching an authorization function to the management interface (via a call to `rpc_mgmt_set_authorization_fn()`), a server can set non-default access to the generic management functions. Although these routines give a management program some control of the server, some of these routines only work locally, so the controlling client must run on the same host as the server.

At the second level, all servers should permit themselves to be managed from remote hosts via the `dced`. The requirements in the server's initialization code are minimal:

1. The server should establish a security state using the `dce_server_sec_begin()` call. This call establishes the server's identity with the RPC runtime such that clients can make authenticated remote procedure calls to it. The call also establishes with the Security Service the server's identity so that it can make authenticated remote procedure calls to other servers.

   Server writers should also give the `dced` (which runs with the host's principal identity) permission to control the server. Since the default is to disable remote control, the server must provide a non-default authorization function that gives the machine principal access. An example of such an authorization function is given in Chapter 5, "Security" on page 163.

2. The server must register as a DCE server using the `dce_server_register()` call. This call fulfills the majority of the server initialization tasks including creating bindings, registering interfaces with the RPC runtime, registering endpoints with `dced`'s endpoint mapper service, and advertising in the name service.

All servers should take these steps to operate correctly in DCE.

Finally, applications can provide application-specific server management. This would typically be done for a persistent server that provides access to some shared resource such as a database. Such a server can provide a set of privileged management operations—such as database maintenance—as a separate application-specific management interface. Such an interface can be accessed by an application management client that can also call the DCE management interfaces. This type of management client is shown in Figure 133 on page 267.

Manager Initialization

Server initialization tasks can typically be divided between essentially generic initialization—creating bindings, establishing security state, exporting to a name service, and listening for calls, among other things—and manager-specific initialization. (Remember that management refers to a set of tasks to control a server while a manager is a server's implementation of a set of operations from one or more interfaces.)

Once the server has called `rpc_server_listen()`, the manager operations may be called asynchronously. The application may, however, need to perform some initialization before any manager operations are performed. For example, the sample storage manager (code example `context_manager.c`) needs to initialize its tables before any storage can be allocated out of them. An application has three choices about manager initialization policy:

1. The server can perform manager initialization before calling `rpc_server_listen()`.

2. The server can have the first instance of a manager operation thread perform manager initialization, using the `pthread_once()` facility. Although initializing everything prior to listening for remote procedure calls is more straight-forward programming, some applications might benefit from this
threaded approach. For example, those operations that do not need the initialization could forgo use of the \texttt{pthread\_once()} facility. This is the approach demonstrated in the sample storage manager.

3. The server can export manager initialization operations as part of its application-specific management interface, and have a management client perform the initialization.

Options 1 and 2 have similar effects and are appropriate for most servers. Option 3 might be appropriate for a persistent server where reinitialization of the running server is a useful operation. Such an operation is a perfect candidate for inclusion in an application-specific management interface for a persistent server.
Appendix A. A Sample Application

This chapter discusses the code for a generic sample application called sample that illustrates the recommended policies. The code is as generic as possible in the sense that it demonstrates things that most servers need to do. This generic server code is contained in the sample_server.c and sample_server.h modules. The application-specific portion consists of a set of simple examples to illustrate various styles of RPC data usage, including: pointers, pipes, and context handles. These illustrations are contained in sample_manager.c (the server side) and sample_client.c (the client side). sample.idl contains a set of sample interface definitions for the illustrated usages.

The source code for sample can be found in the /usr/lpp/dce/examples/sample directory.

Getting Started

See the README file for instructions on building and running the sample code.

A generic Makefile (Makefile) is included that is suitable for building the sample code. Also included are two dcecp TCL scripts, sample_setup.dcecp and sample_unsetup.dcecp that automate the creation and removal of data that the sample program needs (such as CDS entries).

The Generic Server

The generic server implemented by sample_server.c demonstrates a variety of tasks that most servers need to carry out, such as exporting bindings, creating an authentication identity, establishing an ACL manager, and handling asynchronous signals. As much as possible, the bulk of each task is implemented as one or more separate functions. This modularity makes it easier to understand the requirements for coding each task since each function or related set of functions can be studied separately. Also, because the tasks performed are fairly generic, the functions should be reusable in something close to the form presented here by many servers.

The IDL file sample.idl is included mainly to demonstrate the data type declarations used for the ACL manager.

The IDL file sample_db.idl and the ACF file sample_db.acf are required to generate a server-only stub for the database serialization routines used by the ACL manager.

The generic server is then implemented by sample_server.h and sample_server.c.

Note that the server code contained in these files is nearly all generic. In the ACL manager, the only application specific elements are the type of data stored in the object database, declared in sample.idl, and the name and object UUID for the initial object created during ACL manager setup. The export objects operation uses application-specific names and object uuids. The signal catcher thread installs application-specific handling for asynchronous signals, although the actual signal handling code simply causes the listen loop to return and invoke the generic cleanup operations.
Manager and Client

Most of the application-specific server code is contained in \texttt{sample_manager.c}. Since generic client tasks are so simple, the whole client is contained in \texttt{sample_client.c}.

Object Bind Interface

The remote binding interface is implemented in \texttt{sample_bind.c}. The server and client stubs used for registering and binding the \texttt{sample_bind} interface are generated by \texttt{sample_bind.idl} and \texttt{sample_bind.acf}. See \texttt{sample_client.c} and \texttt{sample_server.c} on how this is accomplished.
Appendix B. Another Sample DCE Application: TIMOP

This chapter begins with a discussion of IDL and the interface-definition process. It then introduces the commented client and server source code for TIMOP, a sample DCE application.

Developing a DCE Application

Your first step in coding a DCE application is to define one or more interfaces through which the application’s clients and servers communicate. Interfaces are defined in a declarative Interface Definition Language (IDL), which is similar to C, and then compiled by the IDL compiler.

Interfaces, like most other objects and entities in DCE, are identified to the system by associating each one with a 128-bit Universal Unique Identifier (UUID). Generating a UUID for your application’s interface is the very first step in the IDL process. See [z/OS DCE Application Development Reference] for details on using uuidgen. Run the uuidgen utility with the -i option (to specify an interface UUID) and the -o option (to specify an output file called happy).

In the z/OS UNIX System Services shell, assuming you are using HFS files, the command is as follows:

```
uuidgen -i -o happy.idl
```

In batch, assuming you use PDS files, run UUIDGEN using the following JCL:

```
//JOBNAME JOB (ACCOUNT)...your_job_parameters
//******************************************************************************
// * JCL to run UUIDGEN
//******************************************************************************
//UUIDGEN EXEC UUIDGEN,
// PARMS='-i '-o //DD:UUIDOUT'
//UUIDOUT DD DSN=USERPRFX.IDL(HAPPY),DISP=SHR
```

Running the UUIDGEN generates a skeleton IDL file for HAPPY that contains a new UUID and little else. It is your task to add the rest.

Thus, the simplified development cycle for a DCE application is as follows:

1. Write and compile the IDL file.
2. Write and compile the server and manager implementation code.
3. Write and compile the client implementation code.
4. Link the server object code with the server stub object code, the DCE library, and the definition side-deck for the DCE DLL.
5. Link the client object code with the client stub object code, the DCE library, and the definition side-deck for the DCE DLL.
6. Try running the completed application.

Some of the steps may have to be executed repeatedly.

Figure 134 on page 274 illustrates this process for both the server and the client modules of TIMOP.
Both the server and the client compilation phases are illustrated. As noted in the figure, these can occur on different machines. Note that the interface UUID is generated only once.
The numeric reference keys in the figure indicate the order of the development steps as follows:

1. Run `uuidgen` to get a skeleton IDL file containing a newly generated UUID.

   Complete the file with your interface operation definitions.

2. Compile the completed interface definition file with the IDL compiler.

3. Write the source-code implementation of the interface operations in the various application C-source files and header files,
   and compile them with the header file generated from the IDL compiler.

4. Link the output of the previous step with:
   - The stub module produced by the IDL compiler
   - The DCE library, `libdce`
   - The definition side-deck associated with the DCE DLL.

Of the server files shown in the figure, you are responsible for writing the following:

<table>
<thead>
<tr>
<th>Table 11. Timop Server Source Files</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HFS File Name</strong></td>
</tr>
<tr>
<td>timop.idl</td>
</tr>
<tr>
<td>timop.acf</td>
</tr>
<tr>
<td>timop_manager.c</td>
</tr>
<tr>
<td>timop_server.c</td>
</tr>
<tr>
<td>timop_refmon.c</td>
</tr>
<tr>
<td>timop_server.h</td>
</tr>
</tbody>
</table>

Of the client files shown in the figure, you write the following files:

<table>
<thead>
<tr>
<th>Table 12. Timop Client Source Files</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HFS File Name</strong></td>
</tr>
<tr>
<td>timop_client.h</td>
</tr>
<tr>
<td>timop_client.c</td>
</tr>
<tr>
<td>timop.h</td>
</tr>
<tr>
<td>timop_aux.h</td>
</tr>
</tbody>
</table>

The attribute configuration file, in the figure is optional. Its input to the IDL compiler alters the IDL compiler’s output in various ways.

There is one other important option. The IDL compiler actually operates by first creating C source modules, and then invoking the C compiler to produce its object file output from the C source. Normally, the C source files are then deleted. You can specify that the C source be kept, in which case these source files appear as output too. This possibility is shown in dotted lines in the figure. The IDL compiler’s use of the C compiler is not shown in the diagram.
A server can implement more than one interface. These interfaces must be defined in separate IDL files and compiled separately by the IDL compiler. The interface operations that are implemented in various source code files are then linked with the IDL compiler output.

**What Do Stub Files Do?**

The client and server stub files output by the IDL compiler consist of RPC routines that handle all the mechanical details of packaging and unpackaging data into messages to send over the network. They also handle sending and receiving of the data. This is done in accordance with the specifications you make in the IDL and ACF input files. The IDL file specifications determine how the client/server interaction occurs over the network (the network protocol). The specifications in the ACF file, if it exists, only affect the way the client’s and possibly the server’s application code interacts with their respective stubs.

---

**TIMOP — A Complete Sample Application**

The TIMOP program is a tutorial DCE application sample. It exercises the basic DCE technologies:

- Threads
- RPC
- Security
- Directory
- Time.

TIMOP is not intended to be a model of general application programming techniques. Production applications would typically feature better fault management, use `getopt()`, use a Motif** interface, internationalization, performance optimization, and so on. None of the above is important for this tutorial. TIMOP tries to be straightforward and illustrative of DCE, as much as possible given the complexity of the technologies involved.

To run TIMOP, you must have a DCE cell up and running. This means the systems on which you run the TIMOP client or server must support thread-safe system interfaces (for example, `printf()`). Note that the MVS/ESA library functions are thread-safe. Also, you must be registered as a DCE principal or at least know the password of a principal in your cell to do authenticated RPCs, and be authorized to use certain facilities of the cell (such as the Registry Editor and namespace interface).

**What TIMOP Does**

The TIMOP program has two parts, a client and a server, implemented by the TIMOP client and server processes.

**TIMOP server:** The server offers just one operation. With this operation, clients can learn the span of time it takes the server to calculate the factorial of a random number specified by the client.

**TIMOP client:** The client spawns a number of threads, which make parallel RPC service calls to designated servers. The client prints out the name, invocation order, and time spans reported by the servers, and the random numbers it asked the servers to take the factorial of. It also prints out a total time span that encompasses all the job events at the servers and the sum of the random numbers.

Only UDP is utilized as a least common denominator transport provider. Authentication and integrity-secure RPC are used to make sure the communicated data is correct, and some authorization (named-based, not ACLs) is used. Integrity of the data transferred between the Timop client and server is verified by setting the `protect_level` parameter of the `rpc_binding_set_auth_info()` call to
**rpc_c_protect_level_pkt_integ.**  The Directory Service is used to identify the servers and to mediate the RPC binding between client and server.

All time calculations are done in UTC with $T_0 = 0$ (the Z or Zulu or UTC reference time zone, corresponding to the classical UT GMT time zone). This is not the local wall clock time, because the client and server may be in different time zones.

**Note:** The server and client clocks are all different physical clocks, but they are synchronized by DTS.

**TIMOP and Security**

Because TIMOP uses the Security Service, the TIMOP clients and servers must run as security principals. However, this tutorial example makes minimal use of security. With the code as supplied, the TIMOP client program is run as a principal named `./.../your_cell/tclient`, and the TIMOP server program is run as a principal named `./.../your_cell/tserver`. These names are hardcoded in the TIMOP auxiliary header file, and should be changed to suit your environment prior to compiling the example. For example, both `tclient` and `tserver` could be the person running this program.

The default login contexts used for running TIMOP are `tclient` and `tserver`. That is, when you run the TIMOP server program and the TIMOP client program, log in to DCE as the principal `tserver` and `tclient`, respectively. Also, since the TIMOP server is run with the key of `tserver`, you need to install this key into the key file, `/tmp/tkeyfile`, to which you should have exclusive read and write permission. To do this, see the comments in the TIMOP server header file.

Note that only a simple form of authorization is used, based on principal names, and not ACLs. It is your responsibility to implement an ACL manager using ACL-based authorization, if you require. (Default source code for ACL management is supplied with the DCE, but to have used it in this example would make the code unwieldy.)

**Source Files**

The TIMOP program is built from the following source (either HFS or PDS files):

<table>
<thead>
<tr>
<th>HFS File Name</th>
<th>PDS Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makefile</td>
<td>N/A</td>
<td>The makefile (applies to the z/OS UNIX System Services shell environment)</td>
</tr>
<tr>
<td>timop.idl</td>
<td>USERPRFX.IDL(TIMOP)</td>
<td>The IDL file</td>
</tr>
<tr>
<td>timop.acf</td>
<td>USERPRFX.ACF(TIMOP)</td>
<td>The ACF file</td>
</tr>
<tr>
<td>timop_aux.h</td>
<td>USERPRFX.H(TIMOPAX)</td>
<td>The auxiliary header file</td>
</tr>
<tr>
<td>timop_client.h</td>
<td>USERPRFX.H(TIMOPCL)</td>
<td>The client header file</td>
</tr>
<tr>
<td>timop_client.c</td>
<td>USERPRFX.C(TIMOPCL)</td>
<td>The client program</td>
</tr>
<tr>
<td>timop_server.h</td>
<td>USERPRFX.H(TIMOPSR)</td>
<td>The server header file</td>
</tr>
<tr>
<td>timop_server.c</td>
<td>USERPRFX.C(TIMOPSR)</td>
<td>The server program</td>
</tr>
<tr>
<td>timop_manager.c</td>
<td>USERPRFX.C(TIMOPMR)</td>
<td>The manager routines</td>
</tr>
<tr>
<td>timop_refmon.c</td>
<td>USERPRFX.C(TIMOPRF)</td>
<td>The server reference monitor.</td>
</tr>
</tbody>
</table>
The HFS files are located in the `/usr/lpp/dce/examples/timop` directory. If you cannot find the Timop source files in this directory, consult your DCE administrator to find where they are located on your system.

If you want to run the Timop examples from the PDS files, you must allocate the PDS files listed above and copy the source code from the HFS files into the PDS using the TSO/E OGET command.

Note that you need to change the HFS names of the source files to the shorter PDS names listed in Table 13 on page 277. Also you will have to change the names of the `timop_aux.h`, `timop_server.h` and `timop_client.h` files in the Timop server and client source to the PDS names listed in Table 13 on page 277.

Manager is generic RPC terminology for the part of the server that actually handles the remote operations. In the usual practice as illustrated here, the server program contains the generic routines that start up and initialize the server, and the manager portion contains the application-specific routines that implement the remote operations, among other things.

### Building TIMOP

Prior to building the TIMOP server and client, you need to do the following:

1. Run the IDL compiler with the `no_mepv`, `_no_cpp` and `_keep c_source` options.
2. Define the TIMOP client and TIMOP server principal names to suit your environment in the TIMOP auxiliary header file.

In the Shell: Assuming you use HFS files, to build TIMOP in the shell, adjust `Makefile` to suit your environment, then issue the following command (you'll have to do this separately for every machine architecture you want to use):

   ```
   make -f Makefile
   ```

   Figure 135 on page 279 shows a sample makefile to build `timop`. This makefile assumes you are using HFS files.
IF = timop

IDL = /bin/idl
IDL_FLAGS = -no_cpp -no_mepv -keep c_source
CFLAGS = -DMVS -D_DCE_THREADS -D_OPEN_SYS -Wgz'ro@ot,DLL
LIBS = -l dce /usr/lib/EUVPDLL.x

FROMIDL = $(IF).h $(IF)_cstub.c $(IF)_sstub.c
COBJ = $(IF)_client.o $(IF)_cstub.o
SOBJ = $(IF)_server.o $(IF)_sstub.o $(IF)_manager.o $(IF)_refmon.o

default: $(IF)_client $(IF)_server
$(IF)_client: $(COBJ)
c89 -o $(IF)_client $(COBJ) $(LIBS)
$(IF)_server: $(SOBJ)
c89 -o $(IF)_server $(SOBJ) $(LIBS)

$(COBJ): $(IF).h
$(SOBJ): $(IF).h

$(FROMIDL): $(IDL) $(IF).idl
$(IDL) $(IF).idl $(IDL_FLAGS)

clean:
rm -f $(FROMIDL) g+O#SJ.o

Figure 135. Sample makefile for Building timop

In Batch: See "Writing a Simple Distributed Application on z/OS" on page 29 for examples to compile and link-edit DCE applications in batch.

Installing TIMOP

To run timop, you must first install timop_client and timop_server on the machines you want to use by performing the following steps. (It is assumed you perform these steps in the z/OS UNIX System Services shell.)

1. Add the client and server principals to the Security registry.
2. Create a keyfile to be used by the server.
3. Create a CDS namespace entry for the server to export its binding information to, and for the clients to import its binding information from.
4. Set up the correct permissions on the namespace entry to allow the server to use it, that is, to write to it, correctly.

Assuming the server’s principal name is tserver and the client’s principal name is tclient, as specified in the timop_aux.h file, perform the following steps:

1. Log in to DCE as the cell administrator.
   
   dce_login cell_admin -dce-
   
   You must first log in as the cell administrator to run the following registry operations. Note that the password at your site may be different from that given in the above example. For more information about dce_login, see z/OS DCE User’s Guide

2. Add the server and client principals to the registry, and set up the server’s keyfile.
rgy_edit
Current site is: registry server at /.../your_cell/subsys/dce/sec/master

rgy_edit=> domain principal
Domain changed to: principal
rgy_edit=> add tserver
rgy_edit=> add tclient
rgy_edit=> domain account
Domain changed to: account
rgy_edit=> add tserver -g server-rpc-group -o none -pw tserver_pw -mp -dce-
rgy_edit=> add tclient -g server-rpc-group -o none -pw tclient_pw -mp -dce-
rgy_edit=> ktadd -p tserver -password tserver_pw -f /tmp/tkeyfile
rgy_edit=> quit
bye.

Note that tserver_pw and tclient_pw in the above examples are the passwords you assign to the server and the client respectively. You can substitute any other values you wish, but be sure to remember what the values are as you will need to log in to DCE before running the client and server programs.

Note: You can also use dcecp to set up the server's keyfile.

For additional information on using rgy_edit and dcecp, see z/OS DCE Administration Guide.

The name of the server's keyfile, /tmp/tkeyfile, is specified by the value of the KEYFILE constant in the timop_server.h file; the name you give to the ktadd sub-command must be identical to the value of this constant. Although the timop server does not maintain its own identity, that is, it runs as the principal identity that invoked it, the keyfile is needed for the rpc_server_register_auth_info() call. This call sets up the authentication parameters for clients contacting the timop server.

3. Create the CDS entry that will be used to hold the server's binding information.


cpadd add entry /.:/t_entry

You can substitute any legitimate CDS name in place of t_entry. For further information about rgy_edit see z/OS DCE Administration Guide.

4. Set up the ACL on the entry to allow access to the server.

acledit -e /.:/t_entry -m user:tserver:rwdtc

Note that tserver is the principal name used in the previous steps, and must be identical to the value of the SERVER_PRINC_NAME constant in timop_aux.h. For further information about acl_edit, see z/OS DCE User's Guide.

5. Enable the tserver principal to write to the RPC daemon.

acledit -e /.:/hosts/<your_host>/rpc-daemon -m user:tserver:rwdtc

You have now installed timop.

Running TIMOP

To run timop, you must first start the server, and then invoke one or more clients to perform the timop operation. This is done as follows.
In the Shell: On the machine on which you want to run the server, enter the following:

```
dce_login tserver tserver_pw
timop_server /.:/t_entry/timop
```

You should do this either in background (&), or in a different window from the one in which you intend to run the client, or on a separate terminal.

Note that you have to log in as the `timop` server principal before you can successfully run `timop_server`. This is because `timop_server` assumes that it has been invoked under the correct identity, and does not explicitly acquire its own login context and identity. This has been left out of `timop_server.c` in order to keep `timop` to the essentials.

In the above example, `/.:/t_entry/timop` is the name in the namespace that you want this server to have. That is, the name of the CDS entry to which the server exports, and by means of which it is known to clients. This entry is set up when you run the `rpccp add entry` command described earlier. It can have any name of your choice.

After you have invoked the server, wait until you see the following message:

```
Server /.:/t_entry/timop ready.
```

Now you can invoke the client (either in the same window, if you ran the server in the background, or in a different window). To start up your client, enter the following:

```
dce_login tclient tclient_pw
timop_client /.:/t_entry/timop
```

Note again that you must log in as the `timop` client principal before invoking `timop_client`.

On a successful start up of the `timop` program, `timop_client` will print out results continuously, until you stop it (see "Stopping TIMOP" on page 282).

On multiple machines in the same cell, you can try the following:

```
timop_server /.:/t_entry          # on machine A
```

```
timop_server /.:/x_entry          # on machine B
```

```
timop_server /.:/y_entry          # on machine C
```

```
timop_client /.:/t_entry /.:/x_entry /.:/y_entry # on machine D
```

```
timop_client /.:/y_entry /.:/x_entry /.:/t_entry # on machine E
```

Note that you need to set up `x_entry` and `y_entry` in the namespace first. You can do this as follows:

```
dce_login cell_admin -dce-
rpccp add entry /.:/x_entry
rpccp add entry /.:/y_entry
```

```
acl_edit -e /.:/x_entry -m user:tserver:rwdtc
acl_edit -e /.:/y_entry -m user:tserver:rwdtc
```

Note that if the servers run on machines that are not in the same cell, you must use fully qualified names beginning with `/...`, and not `/.:/` as shown above.

Prior to running the TIMOP client program, log in to DCE as the principal `tclient`. 

---

Appendix B. Another Sample DCE Application: TIMOP 281
In Batch: To start up the TIMOP server, use the following example JCL.

```plaintext
//JOBNAME JOB (ACCOUNT),...your_job_parameters
//***********************************************************************
//* JCL TO STARTUP THE TIMOP SERVER
//*
//***********************************************************************
//TIMOPSR EXEC PGM=TIMOPSR,PARM='POSIX(ON)/ ./t_entry'
//STEPLIB DD DSN=USERPRFX.TIMOP.LOAD,DISP=SHR
```

Figure 136. Example JCL to Start the TIMOP Server

After you have logged into DCE as principal `tclient`, start up the TIMOP client using the example JCL contained in Figure 136.

```plaintext
//JOBNAME JOB (ACCOUNT),...your_job_parameters
//***********************************************************************
//* JCL TO STARTUP THE TIMOP CLIENT
//*
//***********************************************************************
//TIMOPCL EXEC PGM=TIMOPCL,PARM='POSIX(ON)/ ./t_entry'
//STEPLIB DD DSN=USERPRFX.TIMOP.LOAD,DISP=SHR
```

Figure 137. Example JCL to Start the TIMOP Client

Stopping TIMOP

In the Shell, you must kill clients and servers using the `kill` command or `<CNTL+C> C`. In TSO/E, you can CANCEL or PURGE the TIMOP server and client from the SDSF menu.

This leaves server binding information in the endpoint map and namespace, which is normal for persistent servers. The information can always be removed using the CDS Control Program (CDSCP) later on, if necessary.

Further Exercises

After running TIMOP, a good exercise to increase your understanding is to modify TIMOP in various ways, and start writing your own applications. Some suggestions include the following:

- Intentionally introduce some threads race conditions to experiment with the meaning of reentrancy. Then, fix the `asctime()` bug intentionally left in the code.
- Parallelize the client in a different way by using `pthread_exit()` and `pthread_join()` instead of `pthread_cond_signal()` and `pthread_cond_wait()`.
- Receive just one reply from one server, canceling the other outstanding jobs when the first reply arrives.
- Handle server returns from within the listen loop. You have to clean the server binding information from the endpoint map and namespace. You may want to experiment with the `pthread_signal_to_cancel_np()` library routine and the exception-handling interface (the TRY, FINALLY, and ENDTRY constructs). For more information, see Chapter 4, “Threads” on page 149
- Create a namespace service group, instead of a collection of individually named server instances.
• Create Version 1.1 of TIMOP, that contains an additional operation to implement an additive version of the multiplicative factorial job ('n += i' instead of 'n *= i').

• Use context handles and some DTS primitives to return per-client cumulative job times.

• Create a server that supports two managers, each offering a separate implementation of the factorial operation: one implementation remaining the same as the present version, while the new one (accessed by a new object UUID) computes the factorial in decreasing order.

• Working with some other users, make the clients and servers run under several principal identities. Have your security administrator create some extra identities for you to experiment with. The extra identities are also useful in the following exercise.

• Implement an ACL manager for the TIMOP service, and add ACL entries for several principals and groups, and test the ACL manager by running the clients under various principal identities.

• Replace the no-op factorial operation with some operation or operations that would be really useful in your environment. This is the first step in creating your own full-blown DCE application.
The TIMOP Program: A Sample DCE Application

The following subsections present the source code for the TIMOP application.

The TIMOP IDL Source File

```idl
/*
 * timop.idl
 *
 * IDL interface specification for remote time operations.
 */

/* We need explicit handles in timop because our client has multiple (actually, multi-threaded) RPCs bound to multiple explicitly-specified servers. */

timop.idl

[uuid(0cf61668-bb58-11c9-8078-02608c0a03a7),
 version(1.0)]
interface timop
{

    /* DTS timestamps are already in a universal format,
    so are opaque to (the presentation layer of) the RPC
    (16 = sizeof(utc_t)). */
    const small SIZEOF_TIMESTAMP = 16;
    typedef byte timestamp_t[SIZEOF_TIMESTAMP];

    /* Failure value for remote status indications. */
    const long TIMOP_ERR = -1;

    /* Get the time span to do a job (random factorial). */
    [idempotent]
    void timop_getspan(
        [in]    handle_t        handle,
        [in]    long            rand,
        [out]   timestamp_t     timestamp,
        [out]   long            *status_p,
        [in,out] error_status_t *remote_status_p);
}
```

Figure 138. Interface Definition File for TIMOP
The TIMOP ACF Source File

/*
**      timop.acf
**      Attribute configuration file for timop interface.
*/

/* Do all marshalling out-of-line. */
[out_of_line]
interface timop
{
    /* Declare remote_status_p to be a comm_status and
       fault_status parameter. */
    timop_getspan(
        [comm_status,fault_status] remote_status_p);
}

Figure 139. Attribute Configuration File for TIMOP

The TIMOP Auxiliary Header File

/*
**      timop_aux.h
**      Auxiliary info for timop example.
**      There are other ways to do these things, but we're just
**      illustrating the basics here.
*/

/* Principal names for this sample application.
   Change them to suit your environment. */
#define CLIENT_PRINC_NAME   (unsigned_char_t)"/.../mycell/tclient"
#define SERVER_PRINC_NAME   (unsigned_char_t)"/.../mycell/tserver"

/* Well-known object uuid for this sample application. */
#define OBJ_UUID            (unsigned_char_t)"2541af56-43a2-11ca-a9f5-02608c0ffe49"

Figure 140. Auxiliary information for TIMOP
The TIMOP Client Header File

/**
 * timop_client.h
 **
 * Client header file for timop interface.
 */

#define MAX_SERVERS 10 /* single-digit server_num's, 0...9 */
#define CLIENT_NUM -1 /* not equal to any server_num */
#define MAX_RANDOM (10*1000*1000) /* big, to observe threads in action */
#define DO_WORK_OK 0 /* pass */
#define DO_WORK_ERR 1 /* fail */

/* Package up do_work() args in a struct, because
 * pthreads start routines take only one argument. */
typedef struct work_arg {
    int server_num; /* as ordered in arg list */
    unsigned_char_t server_name; /* as named in arg list */
    rpc_binding_handle_t bind_handle; /* binding handle to server */
    idl_long_int rand; /* input to the rpc call */
    int status; /* returned from do_work() */
} work_arg_t;

/* Prototypes for client. */
int main(int _1, char _2[]);
void do_work(work_arg_t _1);
void print_report(unsigned_char_t _1, int _2, utc_t _3, long _4);

Figure 141. Client Header File for TIMOP
The TIMOP Client Source File

/*
 ** timop_client.c
 ** Client program for timop interface.
 */
#pragma runopts(stack(12K,4K,ANY,KEEP))

#include <errno.h>
#include <stdio.h>
#include <locale.h>
#include <dce/rpc.h>
#include <pthread.h>
#include <time.h>
#include <dce/utc.h>
#include "timop.h"
#include "timop_aux.h"
#include "timop_client.h"

long       Rand; /* sum of random numbers */
int        Workers;  /* number of active worker threads */
pthread_mutex_t  Work_mutex; /* guard access to Workers, Rand */
pthread_cond_t  Work_cond; /* condition variable for Workers<0 */

int main(int argc, char *argv[])
{
  int server_num, nservers, ret;
  work_arg_t work_arg[MAX_SERVERS];
  unsigned_char_t server_name[MAX_SERVERS],
                  string_binding, protseq;
  rpc_binding_handle_t bind_handle[MAX_SERVERS];
  unsigned32 status;
  utc_t start_utc, stop_utc, span_utc;
  struct tm time_tm;
  uuid_t obj_uid;
  rpc_ns_handle_t import_context;
  pthread_t thread_id[MAX_SERVERS];

  setlocale(LC_ALL, "");

  /* Check usage and initialize. */
  if (argc < 2 || (nservers = argc-1) > MAX_SERVERS) {
    fprintf(stderr,
           "Usage: %s server_name ...(up to %d server_name's)...\n",
           argv[0], MAX_SERVERS);
    fflush(stderr);
    exit(1);
  }
  for (server_num = 0; server_num < nservers; server_num += 1) {
    server_name[server_num] = (unsigned_char_t *)&argv[1+server_num];
  }

Figure 142 (Part 1 of 8). Client Program for TIMOP Interface
/* Initialize object uuid. */
uuid_from_string(OBJ_UUID, &obj_uuid, &status);
if (status != uuid_s_ok) {
  fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
  fflush(stderr);
  exit(1);
}

/* Import binding info from namespace. */
for (server_num = 0; server_num < nservers; server_num += 1) {
  /* Begin the binding import loop. */
  rpc_ns_binding_import_begin(rpc_c_ns_syntax_dce,
    server_name[server_num], timop_v1_0_c_ifspec,
    &obj_uuid, &import_context, &status);
  if (status != rpc_s_ok) {
    fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
  }
  /* Import bindings one at a time. */
  while (1) {
    rpc_ns_binding_import_next(import_context,
      &bind_handle[server_num], &status);
    if (status != rpc_s_ok) {
      fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
      fflush(stderr);
      exit(1);
    }
    /* Select, say, the first binding over UDP. */
    rpc_binding_to_string_binding(bind_handle[server_num],
      &string_binding, &status);
    if (status != rpc_s_ok) {
      fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
      fflush(stderr);
      exit(1);
    }
    rpc_string_binding_parse(string_binding, NULL,
      &protseq, NULL, NULL, NULL, &status);
    if (status != rpc_s_ok) {
      fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
      fflush(stderr);
      exit(1);
    }
    rpc_string_free(&string_binding, &status);
    ret = strcmp(protseq, "ncadg_ip_udp");
    rpc_string_free(&protseq, &status);
    if (ret == 0) {
      break;
    }
  }
}

Figure 142 (Part 2 of 8). Client Program for TIMOP Interface
/* End the binding import loop. */
rpc_ns_binding_import_done(&import_context, &status);
if (status != rpc_s_ok) {
    fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}

/* Annotate binding handles for security. */
for (server_num = 0; server_num < nservers; server_num += 1) {
    rpc_binding_set_auth_info(bind_handle[server_num],
        SERVER_PRINC_NAME, rpc_c_protect_level_pkt_integ,
        rpc_c_authn_dce_secret, NULL /*default login context*/,
        rpc_c_authz_name, &status);
    if (status != rpc_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        fflush(stderr);
        exit(1);
    }
}

/* Initialize mutex and condition variable. */
ret = pthread_mutex_init(&Work_mutex, pthread_mutexattr_default);
if (ret == -1) {
    fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}
ret = pthread_cond_init(&Work_cond, pthread_condattr_default);
if (ret == -1) {
    fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}

/* Initialize random number generator. */
srand(time(NULL));

/* Initialize work args that are constant throughout main loop. */
for (server_num = 0; server_num < nservers; server_num += 1) {
    work_arg[server_num].server_num = server_num;
    work_arg[server_num].server_name = server_name[server_num];
    work_arg[server_num].bind_handle = bind_handle[server_num];
}

/* Print out the year and date, just once. */
ret = utc_gettime(&start_utc);
if (ret == -1) {
    fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}
ret = utc_gmtime(&time_tm, NULL, NULL, NULL, &start_utc);
if (ret == -1) {
    fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}
fprintf(stdout, "\n%24.24s UTC (Z time zone)\n\n", asctime(&time_tm));
fflush(stdout);

Figure 142 (Part 3 of 8). Client Program for TIMOP Interface
/* Main loop -- never exits -- interrupt to quit. */
while (1) {
    /* Per-loop initialization. We're single-threaded here, so
     * locks and reentrant random number generator unnecessary. */
    Rand = 0;
    Workers = nservers;
    for (server_num = 0; server_num < nservers; server_num++) {
        work_arg[server_num].rand = rand() % MAX_RANDOM;
    }

    /* Get client's start timestamp. */
    ret = utc_gettime(&start_utc);
    if (ret == -1) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        fflush(stderr);
        exit(1);
    }

    /* Spawn a worker thread for each server. */
    for (server_num = 0; server_num < nservers; server_num++) {
        ret = pthread_create(&thread_id[server_num],
                              pthread_attr_default, (void (*)()*)do_work,
                              (void *)&work_arg[server_num]);
        if (ret == -1) {
            fprintf(stderr, "FAULT: %s:%d\n", __FILE__,
                     __LINE__);
            fflush(stderr);
            exit(1);
        }
    }

    /* Reap the worker threads; pthread_cond_wait() semantics
     * requires it to be coded this way. */
    ret = pthread_mutex_lock(&Work_mutex);
    if (ret == -1) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        fflush(stderr);
        exit(1);
    }

    while (Workers != 0) {
        ret = pthread_cond_wait(&Work_cond, &Work_mutex);
        if (ret == -1) {
            fprintf(stderr, "FAULT: %s:%d\n", __FILE__,
                     __LINE__);
            fflush(stderr);
            exit(1);
        }
    }

    ret = pthread_mutex_unlock(&Work_mutex);
    if (ret == -1) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        fflush(stderr);
        exit(1);
    }
}

Figure 142 (Part 4 of 8). Client Program for TIMOP Interface
/* Reclaim storage. */
for (server_num = 0; server_num < nservers; server_num += 1) {
    ret = pthread_detach(&thread_id[server_num]);
    if (ret == -1) {
        fprintf(stderr, "Fault: %s:%d\n", __FILE__,
                __LINE__);
        fflush(stderr);
        exit(1);
    }
}

/* Any failures? */
for (server_num = 0; server_num < nservers; server_num += 1) {
    if (work_arg[server_num].status != DO_WORK_OK) {
        exit(1);
    }
}

/* Get client's stop timestamp. */
ret = utc_gettime(&stop_utc);
if (ret == -1) {
    fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}

/* Calculate the span of client's start and stop timestamps. */
ret = utc_spantime(&span_utc, &start_utc, &stop_utc);
if (ret == -1) {
    fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}

/* Print total results. */
print_report((unsigned_char_t)(client), CLIENT_NUM,
             &span_utc, Rand);
}

/* Not reached. */

void
do_work()
{
    /* Do the work. This is done in parallel threads, so we want it
     * (and the subroutine print_report() it calls) to be reentrant.
     */

    work_arg_t
        *work_arg_p)
    {
        int server_num, *status_p, ret;
        unsigned char_t *server_name;
        rpc_binding_handle_t bind_handle;
        idl_long_int rand, status;
        error_status_t remote_status = rpc_s_ok;
        timestamp_t timestamp;
    }
/* Unpackage the args into local variables. */
server_num = work_arg_p->server_num;
server_name = work_arg_p->server_name;
bind_handle = work_arg_p->bind_handle;
rand = work_arg_p->rand;
status_p = &work_arg_p->status;

/* Do the RPC! */
timop_getspan(bind_handle, rand, timestamp, &status, &remote_status);
if (remote_status != rpc_s_ok) {
    fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
    *status_p = DO_WORK_ERR;
    fflush(stderr);
    pthread_exit(NULL);
} /* Not reached. */

if (status != rand) {
    fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
    *status_p = DO_WORK_ERR;
    fflush(stderr);
    pthread_exit(NULL);
} /* Not reached. */

/* Print report. Not a critical section here because print_report()
   is supposed to be implemented to be reentrant. */
print_report(server_name, server_num, (utc_t *)timestamp, rand);

/* Update Rand and decrement Workers. As implemented, it is a
   critical section, so must be locked. */
ret = pthread_mutex_lock(&Work_mutex);
if (ret == -1) {
    fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}
Workers -= 1;
if (Workers == 0) {
    /* Last worker signals main thread. */
    ret = pthread_cond_signal(&Work_cond);
    if (ret == -1) {
        fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
        fflush(stderr);
        exit(1);
    }
}
Rand += rand;
ret = pthread_mutex_unlock(&Work_mutex);
if (ret == -1) {
    fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}

/* Done. */
*status_p = DO_WORK_OK;
pthread_exit(NULL);
/* Not reached. */

Figure 142 (Part 6 of 8). Client Program for TIMOP Interface
print_report()

Print DTS timestamp interval, to millisecond granularity.
Implemented this way so it is reentrant (assuming all the underlying
OS subroutines it calls are reentrant).
This kind of timestamp manipulation is always messy -- see the
manual for the formats of structures and print-strings we use.

int
print_report(
    unsigned_char_t server_name,
    int server_num,
    utc_t utc_p,
    long rand)
{
    #define LINE_LEN 78
    #define COL1 0
    #define COL2 44
    #define COL3a 47
    #define COL3b 60
    #define COL4 70
    char asctime_buf[26], ascinacc_buf[26],
    time_ns_buf[10], inacc_ns_buf[10],
    report[LINE_LEN+3];
    int inacc_sec, ret;
    long time_ns, inacc_ns;
    struct tm time_tm, inacc_tm;

    /* Print server_name into report. Pad or truncate as necessary. */
    /* Print server_num into report. */
    if (server_num != CLIENT_NUM) {
        sprintf(report+COL2, "%1.1d ", server_num);
    } else {
        sprintf(report+COL2, "%1.1s ", server_num);
    }

    /* Format utc_p and print it into report. */
    ret = utc_gmtime(&time_tm, &time_ns, &inacc_tm, &inacc_ns, utc_p);
    if (ret == -1) {
        fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
        fflush(stderr);
        exit(1);
    }
    memcpy(asctime_buf, asctime(&time_tm), 26);
    /* reentrancy bug! */
    memcpy(ascinacc_buf, asctime(&inacc_tm), 26);
    /* reentrancy bug! */
    sprintf(time_ns_buf, "%9.9d", time_ns);
    sprintf(inacc_ns_buf, "%9.9d", inacc_ns);
    inacc_sec = inacc_tm.tm_yday*24*60*60 + inacc_tm.tm_hour*60 +
                inacc_tm.tm_min*60 + inacc_tm.tm_sec;
    sprintf(report+COL3a, "%8.8s.%3.3sI", asc_tm2buf+11,
            time_ns_buf);
    printf("Figure 142 (Part 7 of 8). Client Program for TIMOP Interface
"
if (inacc_tm.tm_year != -1) {
    sprintf(report+COL3b, "%4.4d.%3.3s ", inacc_sec,
            inacc_ns_buf);
} else {
    sprintf(report+COL3b, "%8.8s ", "infinity");
}

/* Print rand into report. */
if (server_num != CLIENT_NUM) {
    sprintf(report+COL4, "%8d\n", rand);
} else {
    sprintf(report+COL4, "%8d\n\n", rand);
}

/* Output report. */
fprintf(stdout, "%s", report);
fflush(stdout);
return;

Figure 142 (Part 8 of 8). Client Program for TIMOP Interface
The TIMOP Server Header File

/*
** timop_server.h
**
** Server header file for timop interface.
*/

#define NUM_OBJS 1   /* num of objs supported */
#define MAX_CONC_CALLS_PROTSEQ 5  /* max conc calls per protseq */
#define MAX_CONC_CALLS_TOTAL 10  /* max conc calls total */

/* Success/failure for remote procedures. */
#define GETSPAN_OK 0   /* pass */
#define GETSPAN_ERR 1   /* fail */

/* Defines for access control. */
#define GETSPAN_OP 1  /* requested operation */
#define GRANT_ACCESS 0 /* reference monitor success */
#define DENY_ACCESS 1  /* reference monitor failure */
#define IS_AUTHORIZED 0 /* authorization success */
#define NOT_AUTHORIZED 1 /* authorization failure */

/* Server key table for this example. Change name of keyfile to suit your environment, and populate it with the rgyedit subcommand "ktadd -p tserver -pw tserver -f /tmp/tkeyfile". */
#define KEYFILE "/tmp/tkeyfile"
#define KEYTAB "FILE:" ## KEYFILE

/* Prototypes for server. */
int main(int _1, char **_2[]);
void getspan_ep(rpc_binding_handle_t _1, idl_long_int _2, timestamp_t _3,
                idl_long_int *_4, error_status_t _5);
int do_getspan(idl_long_int _1, timestamp_t _2);
int ref_mon(rpc_binding_handle_t _1, int _2);
int is_authorized(unsigned_char_t _1, int _2);
The TIMOP Server Source File

/*
** timop_server.c
**
** Server program for timop interface.
*/
#pragma runopts(stack(12K,4K,ANY,KEEP))

#include <stdio.h>
#include <locale.h>
#include <dce/rpc.h>
#include "timop.h"
#include "timop_aux.h"
#include "timop_server.h"

/* Declare manager EPV. This EPV could be bulk-initialized here, 
   but we do prefer to do it one operation at a time in main(). */
    timop_v1_0_epv_t    manager_epv;

/
*   main()
*
* Get started -- set up server the way we want it, and call listen loop.
*/

int main(int argc, char *argv[]) {
    unsigned_char_t  *server_name;
    rpc_binding_vector_t  *bind_vector_p;
    unsigned32           status;
    int                 i;
    uuid_t              type_uuid, obj_uuid;
    struct {
        unsigned32    count;
        uuid_t        *uuid[NUM_OBJS];
    }          obj_uuid_vec = {NUM_OBJS, {&obj_uuid}};

    setlocale(LC_ALL, "");

    /* Check usage and initialize. */
    if (argc != 2) {
        fprintf(stderr, "Usage: %s namespace_server_name\n", argv[0]);
        fflush(stderr);
        exit(1);
    }
    server_name = (unsigned_char_t *)argv[1];

    /* Initialize manager EPV (just one entry point in this example). */
    manager_epv.timop_getspan = getspan_ep;

    /* Initialize object uuid (just one in this example). */
    uuid_from_string(OBJ_UUID, &obj_uuid, &status);
    if (status != uuid_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        fflush(stderr);
        exit(1);
    }

Figure 143 (Part 1 of 3). Server Program for TIMOP Interface
/* Initialize type uuid (just one in this example). */
uuid_create(&type_uuid, &status);
if (status != uuid_s_ok) {
    fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}

/* Register object/type uuid associations with rpc runtime. */
rpc_object_set_type(&obj_uuid, &type_uuid, &status);
if (status != rpc_s_ok) {
    fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}

/* Register interface/type_uuid/epv associations with rpc runtime. */
rpc_server_register_if(timop_v1_gz'ro@ot_s_ifspec, &type_uuid,
    (rpc_mgr_epv_t)&manager_epv, &status);
if (status != rpc_s_ok) {
    fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}

/* Tell rpc runtime we want to use all supported protocol sequences. */
rpc_server_use_all_protseqs(MAX_CONC_CALLS_PROTSEQ, &status);
if (status != rpc_s_ok) {
    fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}

/* Ask the runtime which binding handle(s) it's going to let us use. */
rpc_server_inq_bindings(&bind_vector_p, &status);
if (status != rpc_s_ok) {
    fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}

/* Register authentication info with rpc runtime. */
rpc_server_register_auth_info(SERVER_PRINC_NAME,
    rpc_c_authn_dce_secret, NULL /*default key retrieval function*/,
    KEYTAB /*server key table for this example*/, &status);
if (status != rpc_s_ok) {
    fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
}

/* Establish server's login context(s), if necessary.
   In this example we just use the default login context,
   so we do NOTHING here. */

---

Figure 143 (Part 2 of 3). Server Program for TIMOP Interface
/* Decide what to do upon server termination. It would be prudent 
to handle signals and decide what to do if the listen loop returns 
(e.g., clean exported information out of endpoint map and namespace, 
something that is not usually done for a persistent server), 
but since this is just an example we don't do those things here. */

/*@ Register binding information with endpoint map. */
rpc_ep_register(timop_v1_0_s_ifspec, bind_vector_p, 
  (uuid_vector_t *)&obj_uuid_vec, 
  (unsigned_char_t *)&"timop server, version 1.0", &status);
if (status != rpc_s_ok) {
  fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
  fflush(stderr);
  exit(1);
}

/*@ Export binding info to the namespace. */
rpc_ns_binding_export(rpc_c_ns_syntax_dce, server_name, 
  timop_v1_0_s_ifspec, bind_vector_p, 
  (uuid_vector_t *)&obj_uuid_vec, &status);
if (status != rpc_s_ok) {
  fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
  fflush(stderr);
  exit(1);
}

/*@ Listen for service requests (semi-infinite loop). */
  fprintf(stdout, "Server %s ready.\n", server_name); 
  fflush(stdout);
  rpc_server_listen(MAX_CONC_CALLS_TOTAL, &status);
  if (status != rpc_s_ok) {
    fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
    fflush(stderr);
    exit(1);
  }

/*@ Returned from listen loop. We haven't arranged for this. */
  fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
  fflush(stderr);
  exit(1);
}

Figure 143 (Part 3 of 3). Server Program for TIMOP Interface

298 DCE Application Development Guide: Introduction and Style
The TIMOP Manager Source File

/ *
** timop_manager.c
**
** Manager routines for timop interface.
*/

#include <stdio.h>
#include <dce/utc.h>
#include "timop.h"
#include "timop_aux.h"
#include "timop_server.h"

/*
 * getspan_ep()
 *
 * Entry point for timop_getspan() operation.
 * Note it is reentrant, so we can have a multi-threaded server.
 */

void
getspan_ep(
    rpc_binding_handle_t bind_handle,
    idl_long_int rand,
    timestamp_t timestamp,
    idl_long_int *status_p,
    error_status_t *remote_status_p)
{
    int ret;

    /* Call reference monitor, to make authorization decision. */
    ret = ref_mon(bind_handle, GETSPAN_OP);
    if (ret == DENY_ACCESS) {
        *status_p = TIMOP_ERR;
        return;
    }

    /* Service the request, i.e., do the actual remote procedure. */
    ret = do_getspan(rand, timestamp);
    if (ret == GETSPAN_ERR) {
        *status_p = TIMOP_ERR;
        return;
    }

    /* Return the input random number as a status value (!= TIMOP_ERR). */
    *status_p = rand;
    /* Return all results to client, and resume listen loop. */
    return;

Figure 144 (Part 1 of 2). Manager Routines for TIMOP Interface
int do_getspan(
    idl_long_int rand, 
    timestamp_t timestamp)
{
    long i;
    volatile long n;
    int ret;
    utc_t start_utc, stop_utc;

    /* Get server's start timestamp. */
    ret = utc_gettime(&start_utc);
    if (ret == -1) {
        fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
        return(GETSPAN_ERR);
    }

    /* Do service (here a random factorial, but could be anything). */
    for (n = i = 1; i <= rand; i += 1) {
        n *= i; /* Burn cpu -- use your imagination. */
    }

    /* Get server's stop timestamp. */
    ret = utc_gettime(&stop_utc);
    if (ret == -1) {
        fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
        return(GETSPAN_ERR);
    }

    /* Calculate the span of server's start and stop timestamps. */
    ret = utc_spantime((utc_t *)timestamp, &start_utc, &stop_utc);
    if (ret == -1) {
        fprintf(stderr, "FAULT: %s:%d
", __FILE__, __LINE__);
        return(GETSPAN_ERR);
    }

    /* Success. */
    return(GETSPAN_OK);
}
/*
**  timop_refmon.c
**  Reference monitor for timop example.
*/

#include <stdio.h>
#include "timop_aux.h"
#include "timop.h"
#include "timop_server.h"

/*
 *  ref_mon()
 *  Reference monitor for timop.
 *  It checks generalities, then calls is_authorized() to check specifics.
 */

int
ref_mon(
    rpc_binding_handle_t bind_handle,
    int requested_op)
{
    int ret;
    rpc_authz_handle_t privs;
    unsigned_char_t *client_princ_name, *server_princ_name;
    unsigned32 protect_level, authn_svc, authz_svc,
    status;

    /* Get client auth info. */
    rpc_binding_inq_auth_client(bind_handle, &privs, &server_princ_name,
        &protect_level, &authn_svc, &authz_svc, &status);
    if (status != rpc_s_ok) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        return(DENY_ACCESS);
    }

    /* Check if selected authn service is acceptable to us. */
    if (authn_svc != rpc_c_authn_dce_secret) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        return(DENY_ACCESS);
    }

    /* Check if selected protection level is acceptable to us. */
    if (protect_level != rpc_c_protect_level_pkt_integ
        && protect_level != rpc_c_protect_level_pkt_privacy) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        return(DENY_ACCESS);
    }

    /* Check if selected authz service is acceptable to us. */
    if (authz_svc != rpc_c_authz_name) {
        fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
        return(DENY_ACCESS);
If rpc_c_authz_dce were being used instead of rpc_c_authz_name, privs would be a PAC (sec_id_pac_t *), not a name as it is here.

client_princ_name = (unsigned_char_t *)privs;

Check if selected server principal name is supported.

if (strcmp(server_princ_name, SERVER_PRINC_NAME) != 0) {
    fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
    return(DENY_ACCESS);
}

Now that things seem generally OK, check the specifics.

ret = is_authorized(client_princ_name, requested_op);
if (ret == NOT_AUTHORIZED) {
    fprintf(stderr, "FAULT: %s:%d\n", __FILE__, __LINE__);
    return(DENY_ACCESS);
}

Cleared all the authorization hurdles -- grant access.

is_authorized()

Check authorization of client to the requested service.

This could be arbitrarily application-specific, but we keep it simple.

A normal application (i.e., one using PACs & ACLs) would be using sec_acl_mgr_is_authorized() instead of this function.

int
is_authorized(unsigned_char_t *client_princ_name, int requested_op)
{
    /* Check if we want to let this client do this operation. */
    if (strcmp(client_princ_name, CLIENT_PRINC_NAME) == 0
        && requested_op == GETSPAN_OP) {
        /* OK, we'll let this access happen. */
        return(IS_AUTHORIZED);
    }

    /* Sorry, Charlie. */
    return(NOT_AUTHORIZED);
}

Figure 145 (Part 2 of 2). Reference Monitor for TIMOP Example
Appendix C. Greet6 ACL Manager Example

The source code listings for the Greet6 ACL Manager example follows:

### Greet6 Server Code

```c
#pragma runopts(stack(12K,4K,ANY,KEEP))

#include <stdio.h>
#include <locale.h>
#include <dce/daclmgrv0.h>
#include <dce/dce_error.h>
#include <pthread.h>
#include <dce/sec_login.h>
#include "greet6.h"
#include "rdaclifv0.h"

globalref rdaclif_v0_0_epv_t rdaclif_v0_0_manager_epv;
globalref greet_v1_0_epv_t greet_v1_0_manager_epv;
#define KEYFILE "/tmp/gkeyfile"
#define KEYTAB "FILE:" # KEYFILE
#define MAX_CONCURRENT_CALLS 5

int main(int argc, char *argv[]) {
    rpc_binding_vector_p_t bvec;
    error_status_t st, error_inq_st;
    ndr_boolean validfamily;
    ndr_char *string_binding;
    int i;
    uuid_vector_t acl_server_obj_uuids;
    sec_acl_mgr_handle_t sec_acl_mgr;
    uuid_t manager_types[1];
    int size_used,num_types;
    char error_text[128];
    pthread_t main_thread = pthread_self();

    /* Declarations for Security */
    sec_passwd_rec_t pwrec;
    boolean32 reset_passwd;
    sec_login_auth_src_t auth_src;
    sec_login_handle_t login_context;
    char dce_login[STR_SZ], passwd[STR_SZ];

    setlocale(LC_ALL, "");
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <PRINCIPAL> <PASSWORD>\n",argv[0]);
        fflush(stderr);
        exit(1);
    }

    strcpy(dce_login,argv[1]);
    strcpy(passwd,argv[2]);

    printf("Establishing login identity with security server...\n");
    fflush(stdout);

    sec_login_setup_identity(dce_login,sec_login_no_flags,&login_context,&st);
```
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot set up login identity: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* Check the passwd.h header file for this structure */

pwrec.key.tagged_union.plain = passwd;
pwrec.key.key_type = sec_passwd_plain;
pwrec.pepper = NULL;
pwrec.version_number = sec_passwd_c_version_none;

sec_login_validate_identity(login_context,&pwrec,&reset_passwd, &auth_src,&st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot validate login identity: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

if (reset_passwd) {
    printf("Password must be changed !\n");
    fflush(stdout);
}

sec_login_set_context(login_context,&st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot set login context: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

fprintf(stdout, "Identity established !\n");
fflush(stdout);

/* Initializing the ACL manager database */

fprintf(stdout, "Initializing ACL database\n");
fflush(stdout);

sec_acl_mgr_configure(sec_acl_mgr_config_create,NULL,&sec_acl_mgr,&st);
fprintf(stdout, "Getting ACL manager types \n");
fflush(stdout);

sec_acl_mgr_get_manager_types(sec_acl_mgr,NULL,sec_acl_type_object,
1,&size_used,&num_types,manager_types,
&st);

printf("Setting Protocol sequence...\n");
fflush(stdout);

/* Calling rpc_server_use_all_protseqs to tell the RPC runtime */
/* to use all supported protocol sequences */

rpc_server_use_all_protseqs(MAX_CONCURRENT_CALLS, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot establish protocol sequences: %s\n", error_text);
    fflush(stderr);
    exit(1);
}
printf("Registering Greet interface...
");
fflush(stdout);

/* Register interface with RPC runtime */

rpc_server_register_if(greet_v1_0_s_ifspec, NULL,
    (rpc_mgr_epv_t) &greet_v1_0_manager_epv, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot register server interface: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

printf("Registering ACL interface...
");
fflush(stdout);

rpc_server_register_if(rdaclif_v0_0_s_ifspec, NULL,
    (rpc_mgr_epv_t)&rdaclif_v0_0_manager_epv, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot register acl interface: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* Get binding handles from the runtime */

rpc_server_inq_bindings(&bvec, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot inquire bindings: %s\n", error_text);
    exit(1);
}

/* Display the bindings */

printf("Server Greet bindings:\n");
fflush(stdout);

for (i=0; i < bvec->count; i++) {
    rpc_binding_to_string_binding(bvec->binding_h[i],&string_binding, &st);
    printf("%s\n", (char *)string_binding);
    fflush(stdout);
    rpc_string_free(&string_binding, &st);
}

/* Register Auth... information with RPC runtime */

printf("Registering Greet Server Auth info...\n");
fflush(stdout);

rpc_server_register_auth_info(dce_login, rpc_c_authn_dce_secret, NULL, KEYTAB, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot register auth - %s\n", error_text);
    exit(1);
}

/* Registering server endpoint with the local Endpoint Map */
printf("Registering Greet interface with EPM...\n");
fflush(stdout);

rpc_ep_register(greet_v1_0_s_ifspec, bvec, NULL,
    "Greet version 1.0 server", &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot register endpoint: %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* Registering the ACL interface with the endpoint mapper */

printf("Registering ACL interface with EPM...\n");
fflush(stdout);

acl_server_obj_uuids.count = 1;
acl_server_obj_uuids.uuid[gz'ro@ot] = &manager_types[gz'ro@ot];
rpc_ep_register(rdaclif_vgz'ro@ot_gz'ro@ot_s_ifspec, bvec, &acl_server_obj_uuids,
    "Greet ACL Manager", &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot register ACL endpoint: %s\n", error_text);
    exit(1);
}

fprintf(stdout, "Exporting server bindings to namespace...\n");
fflush(stdout);

rpc_ns_binding_export(rpc_c_ns_syntax_default,
    (unsigned_char_t*)"/./:<your_dir_name>/greet6",
    greet_v1_0_s_ifspec, bvec, (uuid_vector_t)NULL,&st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot export binding: %s\n", error_text);
    exit(1);
}

/* rdaclif_v0_0_s_ifspec, bvec */

fprintf(stdout, "Exporting ACL bindings to namespace...\n");
fflush(stdout);

rpc_ns_binding_export(rpc_c_ns_syntax_default,
    (unsigned_char_p_t)"/./:<your_dir_name>/greet_acl",
    rdaclif_v0_0_s_ifspec, bvec,
    &acl_server_obj_uuids, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot export ACL binding - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* At this point the server waits for clients */

TRY {

    printf("Server Greet is listening...\n");
    fflush(stdout);


rpc_server_listen(MAX_CONCURRENT_CALLS, &st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot listen - %s\n", error_text);
    fflush(stderr);
    exit(1);
}

CATCH_ALL {
    CLEANUP:
    printf("Unregistering greet interface\n");
    fflush(stdout);

    rpc_server_unregister_if(greet_v1_gzro@ot_s_ifspec, NULL, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot unregister interface: %s\n", error_text);
        fflush(stderr);
    }

    printf("Unregistering ACL interface\n");
    fflush(stdout);

    rpc_server_unregister_if(rdaclif_vgzro@ot_gzro@ot_s_ifspec, NULL, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot unregister interface: %s\n", error_text);
        fflush(stderr);
    }

    printf("Unregistering Greet endpoint\n");
    fflush(stdout);

    rpc_ep_unregister(greet_v1_gzro@ot_s_ifspec, bvec, NULL, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot unregister endpoint: %s\n", error_text);
        fflush(stderr);
    }

    printf("Unregistering ACL endpoint\n");
    fflush(stdout);

    rpc_ep_unregister(rdaclif_vgzro@ot_gzro@ot_s_ifspec, bvec, &acl_server_obj_uuids, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot unregister endpoint: %s\n", error_text);
        fflush(stderr);
    }

    printf("Unexporting Greet handle from the namespace\n");
    fflush(stdout);

    rpc_ns_binding_unexport(rpc_c_ns_syntax_default,"/./<your_dir_name>/greet6",
        greet_v1_gzro@ot_s_ifspec,NULL,&st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &error_inq_st);
        fprintf(stderr, "Cannot unexport Greet handle: %s\n", error_text);
        fflush(stderr);
    }
}
printf("Unexporting ACL handle from the namespace\n");
fflush(stdout);

rpc_ns_binding_unexport(rpc_c_ns_syntax_default,"../<your_dir_name>/greet_acl",
raudif_v0_0_s_ifspec,&acl_server_obj_uuids,&st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &error_inq_st);
    fprintf(stderr, "Cannot unexport ACL handle: %s\n", error_text);
    fflush(stderr);
}
printf("Done!\n");
fflush(stdout);

ENDTRY;
return(gz'ro@ot);
}

Greet6 Manager Code

#include <stdio.h>
#include <dce/id_base.h>
#include <dce/daclmgrv0.h>
#include "greet6.h"

long authorize_client();

/* This is where the RPC call ends up. The code calls the routine
   "authorize_client" to determine if the call can proceed */

void greet_rpc(handle_t h,
           char *client_greeting,
           char *server_reply)
{
    printf("The client says: %s\n",client_greeting);
    fflush(stdout);
    if (authorize_client(h)) {
        fprintf(stdout,"Client is authorized\n");
        fflush(stdout);
        strncpy(server_reply, "Hi Client !", STR_SZ);
    } else {
        printf ("Client is NOT authorized\n");
        fflush (stdout);
        strncpy(server_reply, "You are NOT authorized !", STR_SZ);
    }
}

long authorize_client(rpc_binding_handle_t bh)
{
    rpc_authz_handle_t privs;
    sec_id_pac_t *greetprivs;
    char *server_princ_name;

    /* the code calls the routine "authorize_client" to determine... */

308   DCE Application Development Guide: Introduction and Style
char *str_principal,*str_group,*str_realm;
unsigned32 protect_level,authn_svc,authz_svc,st;
sec_acl_permset_t required_access = sec_acl_perm_unused_00000000;
unsigned32 size_used,num_types;
uuid_t manager_types[1];
char *str_uuid;
int x;

printf("Validating Client\n");
fflush(stdout);

/* Getting the clients PAC from the runtime */

rpc_binding_inq_auth_client(bh, &privs, &server_princ_name,
&protect_level, &authn_svc,
&authz_svc, &st);
if (st != rpc_s_ok) {
    fprintf(stderr, "Client not even close! \n",st);
    fflush(stderr);
} else if (authn_svc != rpc_c_authn_dce_secret) {
    printf("Invalid authentication service.\n");
    flush(stdout);
}
else if (protect_level != rpc_c_protect_level_pkt_integ) {
    printf("Invalid protection level.\n");
    flush(stdout);
}
else if (authz_svc != rpc_c_authz_dce) {
    printf("Invalid authorization level.\n");
    flush(stdout);
}
else if (strcmp(server_princ_name, "/.../Your_Cell_Name/greets") != 0) {
    printf("Invalid server principal name %s.\n",server_princ_name);
    flush(stdout);
} else {
    printf("Client appears to check out !\n");
    flush(stdout);

greetprivs = (sec_id_pac_t *) privs;

printf("Authenticated --> %d \n", greetprivs->authenticated);
fflush(stdout);

uuid_to_string(&greetprivs->principal.uuid,&str_principal,&st);
uuid_to_string(&greetprivs->group.uuid,&str_group,&st);
uuid_to_string(&greetprivs->realm.uuid,&str_realm,&st);

printf("Principal uuid -> %s\n",str_principal);
fflush(stdout);
printf("Group uuid -> %s\n",str_group);
fflush(stdout);
printf("Realm uuid -> %s\n",str_realm);
fflush(stdout);

/* Retrieving the ACL UUID */

sec_acl_mgr_get_manager_types(NULL,NULL,sec_acl_type_object,1,&size_used,
&num_types,manager_types,&st);

uuid_to_string(&(manager_types[0]),&str_uuid,&st);
fprintf(stdout, "Manager uuid %s
",str_uuid);
fflush(stdout);

rpc_ss_enable_allocate();
x = sec_acl_mgr_is_authorized(NULL,required_access,greetprivs,NULL,
&manager_types[0]), NULL,NULL,&st);
rpc_ss_disable_allocate();

return(x);
}
return(0);
}

globaldef greet_v1_0_epv_t greet_v1_0_manager_epv = {greet_rpc};

---

Greet6 secacl Code

#include <dce/daclmgrv0.h>
#include <dce/uuid.h>
#include <stdio.h>

static uuid_t mgr_uuid;
void free_acl();
void get_database_name(uuid_t manager_type, char database_name[],
error_status_t *st);

/* This routine configures the ACL database */
void sec_acl_mgr_configure
{
    sec_acl_mgr_config_t config_info,
unsigned char_p_t db_name,
sec_acl_mgr_handle_t *sec_acl_mgr,
error_status_t *st
}
{
    sec_acl_list_t sec_acl_list;
int size_used,num_types;
uuid_t manager_types[i];
char* dummy_name[10];
uuid_t dummy_uuid;
char* database_name[1024];
FILE *fp;

fprintf(stdout, "Inside sec_acl_mgr_configure\n");
fflush(stdout);

*sec_acl_mgr = NULL;

/* Setting the ACL manager type uuid. This is so the ACL can be
uniquely identified from anywhere */

uuid_from_string("00668186-B089-1B55-A97A-1005AA89D46",&mgr_uuid,&st);

/* Retrieves the ACL file name based on the uuid */

310 DCE Application Development Guide: Introduction and Style
get_database_name(&mgr_uuid, database_name, st);

if (config_info == sec_acl_mgr_config_create) {
    /* If our database file already exists, then do not initialize it! */
    if ((fp = fopen(database_name,"r")) != NULL) {
        *st = sec_acl_mgr_file_open_error;
        fclose(fp);
        return;
    }
    /* Initializing the ACL */
    sec_acl_list.num_acls = 1;
    sec_acl_list.sec_acls[0]->sec_acl_manager_type = mgr_uuid;
    sec_acl_list.sec_acls[0]->num_entries = 0;
    sec_acl_list.sec_acls[0]->sec_acl_entries = NULL;
    /* Creating a dummy realm name and dummy realm uuid */
    /* This can be reset to an actual name and uuid when you */
    /* run the acl_edit command. */
    strcpy(dummy_name,"greet");
    uuid_create_nil(&dummy_uuid,st);
    sec_acl_list.sec_acls[0]->default_realm.name = (char *) calloc(1,strlen(dummy_name) + 1);
    strcpy(sec_acl_list.sec_acls[0]->default_realm.name,dummy_name);
    sec_acl_list.sec_acls[0]->default_realm.uuid = dummy_uuid;
    /* Calling "sec_acl_mgr_replace" to write the info to the file */
    sec_acl_mgr_replace(NULL,db_name,&mgr_uuid,sec_acl_type_object,&sec_acl_list,st);
    free(sec_acl_list.sec_acls[0]->default_realm.name);
}

*st = error_status_ok;
fclose(fp);

fprintf(stdout, "Leaving sec_acl_mgr_configure\n");
fflush(stdout);
}

/* This routine finds the total permissions a user has on the object. */
/* If the principal has a user entry and a group entry in the ACL, then */
/* the permissions are combined. */

void sec_acl_mgr_get_access(
    sec_acl_mgr_handle_t sec_acl_mgr,
    sec_id_pac_t *accessor_info,
    sec_acl_key_t sec_acl_key,
    uuid_t *manager_type,
    sec_id_t *user_obj,
    sec_id_t *group_obj,
    sec_acl_permset_t *net_rights,
    error_status_t *st

) {
    sec_acl_list_t *sec_acl_list;
    sec_acl_t *sec_acl_p;
}
int i;  /* For traversing entry list. */

fprintf(stdout, "Inside sec_acl_manager get_access...\n");
fflush(stdout);

sec_acl_mgr_lookup(sec_acl_mgr, sec_acl_key, manager_type,
sec_acl_type_object, &sec_acl_list, st);
if (*st != error_status_ok) {
    fprintf(stderr, "Leaving sec_acl_manager get_access...\n");
    fflush(stderr);
    return;
}

sec_acl_p = sec_acl_list->sec_acls[0];
*net_rights = 0;
for (i = 0; i < sec_acl_p->num_entries; i++) {
    switch(sec_acl_p->sec_acl_entries[i].entry_info.entry_type) {
    case sec_acl_e_type_user:
        if (uuid_equal(&(accessor_info->principal.uuid),
                       &sec_acl_p->sec_acl_entries[i].entry_info.tagged_union.id.uuid), st))
            *net_rights = (*net_rights | sec_acl_p->sec_acl_entries[i].perms);
        break;
    case sec_acl_e_type_group:
        if (uuid_equal(&(accessor_info->group.uuid),
                       &sec_acl_p->sec_acl_entries[i].entry_info.tagged_union.id.uuid), st))
            *net_rights = (*net_rights | sec_acl_p->sec_acl_entries[i].perms);
        break;
    default:
        break;
    }
}
*st = error_status_ok;

fprintf(stdout, "Leaving sec_acl_manager get_access...\n");
fflush(stdout);
return;
}

/* This routine returns all of the available manager types.  
   Since we only have a single manager type, that is all we return */

void sec_acl_mgr_get_manager_types
(
    sec_acl_mgr_handle_t sec_acl_mgr,
    sec_acl_key_t sec_acl_key,
    sec_acl_type_t sec_acl_type,
    unsigned32 size_avail,
    unsigned32 *size_used,
    unsigned32 *num_types,
    uuid_t *manager_types[],
    error_status_t *st
)
{
    fprintf(stdout, "Inside sec_acl_mgr_get_manager_types\n");
flush(stdout);

*num_types = 1;

if (size_avail < 1)
    *size_used = 0;
else {
    *size_used = 1;
    manager_types[0] = mgr_uuid;
}

*st = error_status_ok;

fprintf(stdout, "Leaving sec_acl_mgr_get_manager_types\n");
fflush(stdout);

#define NUM_PSTRS 8

/* These permissions are from the dacl_manager example. A new permission "g" has been added for this Greet example */

static sec_acl_printstring_t hardcoded_printstrings[] = {
    { "g", "greet", sec_acl_perm_unused | sec_acl_perm_owner | sec_acl_perm_read | sec_acl_perm_write | sec_acl_perm_execute | sec_acl_perm_insert | sec_acl_perm_delete | sec_acl_perm_test | sec_acl_perm_unused | sec_acl_perm_test };

static sec_acl_printstring_t hardcoded_manager_info = {
    "acl_test_server", "Sample ACL manager.",
    (sec_acl_perm_owner | sec_acl_perm_read | sec_acl_perm_write | sec_acl_perm_execute | sec_acl_perm_insert | sec_acl_perm_delete | sec_acl_perm_test | sec_acl_perm_unused | sec_acl_perm_test |

void sec_acl_mgr_get_printstring
(
    sec_acl_mgr_handle_t sec_acl_mgr,
    uuid_t *manager_type,
    unsigned32 size_avail,
    uuid_t *manager_type_chain,
    sec_acl_printstring_t *manager_info,
    boolean32 *tokenize,
    unsigned32 *total_num_printstrings,
    unsigned32 *size_used,
    sec_acl_printstring_t printstrings[],
    error_status_t *st
)
{
    int i;
    error_status_t err_st;
    char *uuid1, *uuid2;

    fprintf(stdout, "Inside sec_acl_mgr_get_printstring...\n");
    fflush(stdout);
if (size_avail < NUM_PSTRS)
    *size_used = size_avail;
else
    *size_used = NUM_PSTRS;

uuid_create_nil(manager_type_chain, &err_st);
*tokenize = false;
*total_num_printstrings = NUM_PSTRS;

strcpy(manager_info->printstring, hardcoded_manager_info.printstring);
strcpy(manager_info->helpstring, hardcoded_manager_info.helpstring);
manager_info->permissions = hardcoded_manager_info.permissions;

if (!uuid_equal(&mgr_uuid, manager_type, &err_st)) {
    *st = sec_acl_unknown_manager_type;
    fprintf(stderr, "Unknown manager type.\n");
    fflush(stderr);
    fprintf(stderr, "Leaving sec_acl_mgr_get_printstring.\n");
    fflush(stderr);
    return;
}
else
    for (i = gotoroot; i < *size_used; i++) {
        strcpy(printstrings[i].printstring, hardcoded_printstrings[i].printstring);
        strcpy(printstrings[i].helpstring, hardcoded_printstrings[i].helpstring);
        printstrings[i].permissions = hardcoded_printstrings[i].permissions;
    }

*st = error_status_ok;

fprintf(stdout, "Leaving sec_acl_mgr_get_printstring.\n");
fflush(stdout);

}
/* This routine checks to see if a user is authorized. Currently, it ONLY checks local user and group entries. */

boolean32 sec_acl_mgr_is_authorized
(
    sec_acl_mgr_handle_t sec_acl_mgr,
    sec_acl_permset_t desired_access,
    sec_id_pac_t *accessor_info,
    sec_acl_key_t sec_acl_key,
    uuid_t *manager_type,
    sec_id_t *user_obj,
    sec_id_t *group_obj,
    error_status_t st
)
{
    sec_acl_list_t sec_acl_list;
    sec_acl_t sec_acl_p;
    int i; /* For traversing entry list. */
    sec_acl_permset_t granted;

    fprintf(stdout, "Inside sec_acl_mgr_is_authorized...\n");
    fflush(stdout);

    sec_acl_mgr_lookup(sec_acl_mgr, sec_acl_key, manager_type,
                       sec_acl_type_object, &sec_acl_list, st);
    if (st != error_status_ok)
        return false;

    sec_acl_p = sec_acl_list->sec_acls[0];
    for (i = 0; i < sec_acl_p->num_entries; i++) {
        switch(sec_acl_p->sec_acl_entries[i].entry_info.entry_type) {
        case sec_acl_e_type_user:
            printf("Validating user uuid\n");
            if (uuid_equal(&(accessor_info->principal.uuid),
                           &sec_acl_p->sec_acl_entries[i].entry_info/tagged_union.id.uuid), st))
                if (grant_access(sec_acl_p->sec_acl_entries[i].perms, desired_access, &granted)) {
                    fprintf(stdout, "Leaving sec_acl_mgr_is_authorized... \n");
                    fflush(stdout);
                    return true;
                }
            break;

        case sec_acl_e_type_group:

            printf(stdout, "Validating group uuid\n");
            fflush(stdout);

            if (uuid_equal(&(accessor_info->group.uuid),
                           &sec_acl_p->sec_acl_entries[i].entry_info/tagged_union.id.uuid), st))
                if (grant_access(sec_acl_p->sec_acl_entries[i].perms, desired_access, &granted)) {
                    fprintf(stdout, "Leaving sec_acl_mgr_is_authorized... \n");
                    fflush(stdout);
                    return true;
                }
            break;

        Appendix C. Greet6 ACL Manager Example
default:
    break;
}

fprintf(stdout, "Leaving sec_acl_mgr_is_authorized...
");
fflush(stdout);

return false;
}

/* This routine retrieves the ACL from the database */

void sec_acl_mgr_lookup
(
    sec_acl_mgr_handle_t sec_acl_mgr,
    sec_acl_key_t sec_acl_key,
    uuid_t manager_type,
    sec_acl_type_t sec_acl_type,
    sec_acl_list_t sec_acl_list,
    error_status_t st
)
{
    FILE *fp;
    sec_acl_t sec_acl;
    sec_acl_entry_t sec_acl_entry;
    int i;
    int num_entries;
    char str_uuid1[40], str_uuid2[40], name[124];
    char database_name[124];
    sec_acl_permset_t perms;
    int entry_type;
    char *str_uuid;
    error_status_t err_st;

    fprintf(stdout, "Inside sec_acl_mgr_lookup...
");
    fflush(stdout);

    *sec_acl_list = (sec_acl_list_t *) rpc_ss_allocate(sizeof(sec_acl_list_t));

    (*sec_acl_list)->num_acls = 1;
    (*sec_acl_list)->sec_acls[0] = (sec_acl_t *) rpc_ss_allocate(sizeof(sec_acl_t));

    sec_acl = (*sec_acl_list)->sec_acls[0];

    get_database_name(manager_type, database_name, st);

    fprintf(stdout, "Opening %s
", database_name);
    fflush(stdout);

    fp = fopen(database_name, "r, recfm=*");

    if (fp == NULL) {
        *st = sec_acl_cant_allocate_memory;

        fprintf(stderr, "Can't open %s
", database_name);
        fflush(stderr);
        fprintf(stderr, "Leaving sec_acl_mgr_lookup...
");
        fflush(stderr);
    } else { /* Read the ACL */

        /* Read the database file */

        /* Process the ACL entries */

        /* Close the database file */

        /* Free the allocated memory */

    }

    /* Free the allocated resources */

    /* Return the error status */

    return st;
}
return;
}

fprintf(stderr, "Getting %s database info\n", database_name);
fflush(stderr);

fscanf(fp,"%s %s %s %d", name, str_uuid1, str_uuid2, &num_entries);

fprintf(stderr, "1 %s\n2 %s\n3 %s\n4 %d\n", name, str_uuid1, str_uuid2, num_entries);
fflush(stderr);

sec_acl->default_realm.name = (char *) rpc_ss_allocate(strlen(name));
strcpy(sec_acl->default_realm.name, name);

uuid_from_string(str_uuid1, &(sec_acl->default_realm.uuid), st);
uuid_from_string(str_uuid2, &(sec_acl->sec_acl_manager_type), st);
sec_acl->num_entries = num_entries;

sec_acl->sec_acl_entries = (sec_acl_entry_t *) rpc_ss_allocate(num_entries * sizeof(sec_acl_entry_t));

fprintf(stderr, "\nUsers...\n" );
fflush(stderr);

for (i = 0; i < num_entries; i++) {
    sec_acl_entry = &(sec_acl->sec_acl_entries[i]);
    fscanf(fp,"%d %d %s %s\n", &perms, &entry_type, name, str_uuid1);
    fprintf(stderr, "%d %d %s %s\n", perms, entry_type, name, str_uuid1);
    fflush(stderr);

    sec_acl_entry->perms = perms;
    sec_acl_entry->entry_info.entry_type = (sec_acl_entry_type_t) entry_type;
    sec_acl_entry->entry_info.tagged_union.id.name = (char *) rpc_ss_allocate(strlen(name));
    strcpy(sec_acl_entry->entry_info.tagged_union.id.name, name);
    uuid_from_string(str_uuid1, &(sec_acl_entry->entry_info.tagged_union.id.uuid), st);
}

fclose(fp);

*st = error_status_ok;

fprintf(stderr,"Leaving sec_acl_mgr_lookup...\n" );
fflush(stderr);

return;

/* This routine dumps the ACL to the file */

void sec_acl_mgr_replace
(
    sec_acl_mgr_handle_t sec_acl_mgr,
    sec_acl_key_t sec_acl_key,
    uuid_t *manager_type,
    sec_acl_type_t sec_acl_type,
    sec_acl_list_t *sec_acl_list,
}
error_status_t *st
{
    FILE *fp;
    sec_acl_t *sec_acl;
    sec_acl_entry_t *sec_acl_entry;
    int i;
    int num_entries;
    char *str_uuid1,*str_uuid2;
    char database_name[1024];

    fprintf(stdout, "Inside sec_acl_mgr_replace\n");
    fflush(stdout);

    sec_acl = sec_acl_list->sec_acls[0];
    get_database_name(manager_type,database_name,st);
    fp = fopen(database_name,"w,recfm=");
    if (fp == NULL) {  
        *st = sec_acl_cant_allocate_memory;
        fprintf(stderr, "Can't open %s\n",database_name);
        fflush(stderr);
        fprintf(stderr, "Leaving sec_acl_mgr_replace...
");
        fflush(stderr);
        return;
    }

    uuid_to_string(&(sec_acl->default_realm.uuid),&str_uuid1,st);
    uuid_to_string(&(sec_acl->sec_acl_manager_type),&str_uuid2,st);
    fprintf(stdout, "%s %s %s %d\n",sec_acl->default_realm.name,
            str_uuid1,
            str_uuid2,
            sec_acl->num_entries);
    fflush(stdout);

    fprintf(fp,"%s
%s
%s
%d
",sec_acl->default_realm.name,
            str_uuid1,
            str_uuid2,
            sec_acl->num_entries);
    for (i=0;i<sec_acl->num_entries;i++) {
        sec_acl_entry = &(sec_acl->sec_acl_entries[i]);
        uuid_to_string(&(sec_acl_entry->entry_info.tagged_union.id.uuid),&str_uuid1,st);
        fprintf(stdout, "%d %d %s %s\n",sec_acl_entry->perms,
                sec_acl_entry->entry_info.entry_type,
                sec_acl_entry->entry_info.tagged_union.id.name,
                str_uuid1);
        fflush(stdout);

        fprintf(fp,"%d %d %s %s
",sec_acl_entry->perms,
                sec_acl_entry->entry_info.entry_type,
                sec_acl_entry->entry_info.tagged_union.id.name,
                str_uuid1);
    }
fclose(fp);

fprintf(stdout, "Leaving sec_acl_mgr_replace\n");
fflush(stdout);

*st = error_status_ok;
}

void get_database_name(uuid_t *manager_type, char database_name[], error_status_t *st) {
    if (uuid_equal(manager_type, &mgr_uuid, st))
        strcpy(database_name, "GREETACL");
}

Greet6 rdacl Code

#include <stdio.h>
#include <string.h>
#include <dce/uuid.h>
#include <dce/rdaclifv0.h>
#include <dce/daclmgrv0.h>

void rdacl_lookup(handle_t h, sec_acl_component_name_t component_name,
                  uuid_t *manager_type_p, sec_acl_type_t sec_acl_type,
                  sec_acl_result_t *sec_acl_result_p) {
    error_status_t st;
    fprintf(stdout, "Inside rdacl_lookup...\n");
    fflush(stdout);
    fprintf(stdout, "This is the component name %s\n", component_name);
    fflush(stdout);

    sec_acl_mgr_lookup(NULL, (sec_acl_key_t) component_name,
                       manager_type_p, sec_acl_type,
                       &sec_acl_result_p->tagged_union.sec_acl_list, &st);
    if (st == error_status_ok)
        sec_acl_result_p->st = error_status_ok;
    else
        sec_acl_result_p->st = st;

    fprintf(stdout, "Leaving rdacl_lookup\n");
    fflush(stdout);
}

void rdacl_replace(handle_t h, sec_acl_component_name_t component_name,
                   uuid_t *manager_type_p, sec_acl_type_t sec_acl_type,
                   sec_acl_list_t *sec_acl_list_p, st_p) {
    error_status_t st;
    fprintf(stdout, "Inside rdacl_replace...\n");
    fflush(stdout);
    fprintf(stdout, "This is the component name %s\n", component_name);
    fflush(stdout);

    sec_acl_mgr_replace(NULL, (sec_acl_key_t) component_name,
                        manager_type_p, sec_acl_type,
                        &sec_acl_list_p->tagged_union.sec_acl_list, &st);
    if (st == error_status_ok)
        sec_acl_list_p->st = error_status_ok;
    else
        sec_acl_list_p->st = st;

    fprintf(stdout, "Leaving rdacl_replace\n");
    fflush(stdout);
}
error_status_t *st_p;
{
    fprintf(stdout, "Inside rdacl_lookup...
");
    fflush(stdout);

    sec_acl_mgr_replace(NULL, (sec_acl_key_t) component_name,
                        manager_type_p, sec_acl_type, sec_acl_list_p, st_p);

    fprintf(stdout, "Leaving rdacl_replace\n");
    fflush(stdout);
}

boolean32 rdacl_test_access(h, component_name, manager_type_p, desired_permset, st_p)
handle_t h;
sec_acl_component_name_t component_name;
uuid_t *manager_type_p;
sec_acl_permset_t desired_permset;
error_status_t *st_p;
{
    error_status_t st;
    sec_id_pac_t unauth_pac;
    sec_id_t *user_obj = NULL;
    sec_id_t *group_obj = NULL;
    rpc_authz_handle_t privs;
    unsigned_char_p_t server_princ_name;
    unsigned32 authn_level;
    unsigned32 authn_svc;
    unsigned32 authz_svc;
    unsigned_char_t dummy_name = '\0';
    boolean32 is_auth;

    fprintf(stdout, "Entered rdacl_test_access\n");
    fflush(stdout);

    /*
    if (! uuid_equal(&mgr_uuid, manager_type_p, &st)) {
        *st_p = sec_acl_unknown_manager_type;
        return false;
    }
    else
     */
    *st_p = error_status_ok;

    /* Inquiring the runtime as to who called us */
    rpc_binding_inq_auth_client(h, &privs, &server_princ_name, &authn_level,
                                 &authn_svc, &authz_svc, st_p);
    if (*st_p != error_status_ok) {
        fprintf(stderr, "Leaving rdacl_test_access\n");
        fflush(stderr);
        return false;
    }
    else {
        /* May need to generate an unauthenticated (dummy) PAC */
        if (authz_svc != rpc_c_authz_dce) {

            /*
            */

        }
    }
}
unauth_pac.pac_type = sec_id_pac_format_v1;
unauth_pac.authenticated = false;
uuid_create_nil(&unauth_pac.realm.uuid, &st);
unauth_pac.principal.uuid = unauth_pac.realm.uuid;
unauth_pac.group.uuid = unauth_pac.realm.uuid;
unauth_pac.realm.name = &dummy_name;
unauth_pac.principal.name = &dummy_name;
unauth_pac.group.name = &dummy_name;
unauth_pac.num_groups = 0;
unauth_pac.num_foreign_groups = 0;
unauth_pac.groups = NULL;
unauth_pac.foreign_groups = NULL;
privs = (rpc_authz_handle_t) &unauth_pac;

is_auth = sec_acl_mgr_is_authorized(NULL, desired_permset,
   (sec_id_pac_t *) privs,
   (sec_acl_key_t) component_name,
   manager_type_p, user_obj,
   group_obj, st_p);

fprintf(stdout, "Leaving rdacl_test_access\n");
fflush(stdout);
return is_auth;
}

boolean32 rdacl_test_access_on_behalf(h, component_name, manager_type_p, subject_p, desired_permset, st_p)

handle_t  h;
sec_acl_component_name_t  component_name;
uuid_t  *manager_type_p;
sec_id_pac_t  *subject_p;
sec_acl_permset_t  desired_permset;
error_status_t  *st_p;
{
  error_status_t  st;
  sec_id_pac_t  unauth_pac;
  sec_id_t  *user_obj = NULL;
  sec_id_t  *group_obj = NULL;
  rpc_authz_handle_t  privs;
  unsigned_char_p_t  server_princ_name;
  unsigned32  autn_level;
  unsigned32  authn_svc;
  unsigned32  authz_svc;
  unsigned char_t  dummy_name = '\0';
  boolean32  isauth;

  /*
   * if (! uuid_equal(&mgr_uuid, manager_type_p, &st))
   *   *st_p = sec_acl_unknown_manager_type;
   * else
   */
  *st_p = error_status_ok;

  /* Inquire the runtime as to who called us */
  rpc_binding_inq_auth_client(h, &privs, &server_princ_name, &autn_level,
   &authn_svc, &authz_svc, st_p);
  if (*st_p != error_status_ok)
return false;
else {

/* May need to generate an unauthenticated (dummy) PAC */

if (authz_svc != rpc_c_authz_dce) {
  unauth_pac.pac_type = sec_id_pac_format_v1;
  unauth_pac.authenticated = false;
  uuid_create_nil(&unauth_pac.realm.uuid, &st);
  unauth_pac.principal.uuid = unauth_pac.realm.uuid;
  unauth_pac.group.uuid = unauth_pac.realm.uuid;
  unauth_pac.realm.name = &dummy_name;
  unauth_pac.principal.name = &dummy_name;
  unauth_pac.group.name = &dummy_name;
  unauth_pac.num_groups = 0;
  unauth_pac.num_foreign_groups = 0;
  unauth_pac.groups = NULL;
  unauth_pac.foreign_groups = NULL;
  privs = (rpc_authz_handle_t) &unauth_pac;
}

return( (sec_acl_mgr_is_authorized (NULL, desired_permset,
  (sec_id_pac_t) privs,
  (sec_acl_key_t) component_name,
  manager_type_p, user_obj,
  group_obj, st_p))
  && (sec_acl_mgr_is_authorized (NULL, desired_permset,
  (sec_id_pac_t) subject_p,
  (sec_acl_key_t) component_name,
  manager_type_p, user_obj,
  group_obj, st_p)) );
}

void rdacl_get_manager_types(h, component_name, acl_type, size_avail, size_used_p, num_types_p,
  manager_types, st_p)

handle_t h;
sec_acl_component_name_t component_name;
sec_acl_type_t acl_type;
unsigned32 size_avail;
unsigned32 *size_used_p;
unsigned32 *num_types_p;
uuid_t *manager_types;
error_status_t *st_p;

{  
fprintf(stdout, "Entering rdacl_get_manager_types...\n");
fflush(stdout);

  sec_acl_mgr_get_manager_types(h,component_name,acl_type,size_avail,
    size_used_p,num_types_p,manager_types,st_p);
  fprintf(stdout, "Leaving rdacl_get_manager_types\n");
  fflush(stdout);
}

/* This one is important because dcecp calls this routine first to get the 
   manager type and the POSIX semantics */

void rdacl_get_mgr_types_semantics(h, component_name, acl_type,
size_avail, size_used_p, num_types_p,
manager_types, posix_semantics, st_p)

handle_t h;
sec_acl_component_name_t component_name;
sec_acl_type_t acl_type;
unsigned32 size_avail;
unsigned32 *size_used_p;
unsigned32 *num_types_p;
uuid_t *manager_types;
sec_acl_posix_semantics_t posix_semantics[];
error_status_t *st_p;
{
    uuid_t manager_types_array[1];
    unsigned32 size_used, num_types;

    fprintf(stdout, "Inside rdacl_get_mgr_types_semantics...%d\n", size_avail);
    fflush(stdout);

    sec_acl_mgr_get_manager_types(h, component_name, acl_type, 1,
    &size_used, &num_types, manager_types_array, st_p);

    *num_types_p = 1;
    if (size_avail < 1)
        *size_used_p = 0;
    else {
        *size_used_p = 1;
        manager_types[0] = manager_types_array[0];
        posix_semantics[0] = sec_acl_posix_no_semantics;
    }

    *st_p = error_status_ok;
    fprintf(stdout, "Leaving rdacl_get_mgr_types_semantics.\n");
    fflush(stdout);
}

void rdacl_get_printstring(h, manager_type_p, size_avail, manager_type_chain,
    manager_info, tokenize_p, total_num_printstrings_p,
    size_used_p, printstrings, st_p)

handle_t h;
uuid_t *manager_type_p;
unsigned32 size_avail;
uuid_t *manager_type_chain;
sec_acl_printstring_t *manager_info;
boolean32 *tokenize_p;
unsigned32 *total_num_printstrings_p;
unsigned32 *size_used_p;
sec_acl_printstring_t printstrings[];
error_status_t *st_p;
{
    fprintf(stdout, "Inside rdacl_get_printstring...\n");
    fflush(stdout);

    sec_acl_mgr_get_printstring(h, manager_type_p, size_avail, manager_type_chain,
        manager_info, tokenize_p, total_num_printstrings_p,
        size_used_p, printstrings, st_p);
    fprintf(stdout, "Leaving rdacl_get_printstring\n");
void rdacl_get_referral(h, component_name, manager_type_p, sec_acl_type, towers_p, st_p)
{
    fprintf(stdout, "Inside rdacl_get_referral...\n");
    fflush(stdout);
    *st_p = sec_acl_not_implemented;
    fprintf(stdout, "Leaving rdacl_get_referral...\n");
    fflush(stdout);
}

void rdacl_get_access(h, component_name, manager_type, net_rights, st_p)
{
    rpc_authz_handle_t privs;
    sec_id_pac_t *PAC;
    char *server_princ_name;
    unsigned32 protect_level, authn_svc, authz_svc, st;

    fprintf(stdout, "Entering rdacl_get_access...\n");
    fflush(stdout);
    rpc_binding_inq_auth_client(h, &privs, &server_princ_name, &protect_level, &authn_svc, &authz_svc, &st);
    PAC = (sec_id_pac_t *) privs;

    sec_acl_mgr_get_access(h, PAC, component_name, manager_type, NULL, NULL, net_rights, st_p);
    fprintf(stdout, "Leaving rdacl_get_access\n");
    fflush(stdout);
}

globaldef rdaclif_v0_0_epv_t rdaclif_v0_0_manager_epv = {
    rdacl_lookup,
    rdacl_replace,
    rdacl_get_access,
    rdacl_test_access,
    rdacl_test_access_on_behalf,
    rdacl_get_manager_types,
    rdacl_get_printstring,
    rdacl_get_referral,
Greet6 Client Code

```c
#include <stdio.h>
#include <locale.h>
#include <dce/dce_error.h>
#include <dce/sec_login.h>
#include "greet6.h"

int main(int argc, char argv[])
{
    rpc_binding_handle_t h;
    dce_error_string_t error_text;
    error_status_t st, st1;
    idl_char *string_binding;
    int i, MAX_PASS;
    char reply[STR_SZ];
    char server_principal[STR_SZ];
    char *member;
    rpc_ns_import_handle_t import_context;

    /* Declarations for Security */
    sec_passwd_rec_t pwrec;
    boolean32 reset_passwd;
    sec_login_auth_src_t auth_src;
    sec_login_handle_t login_context;
    char dce_login[STR_SZ], passwd[STR_SZ];

    setlocale(LC_ALL, "");

    if (argc != 5) {
        fprintf(stderr, "Usage: %s <CLIENT PRINCIPAL> <PASSWD> <SERVER PRINCIPAL NAME> <MAX_PASS>\n", argv[0]);
        fflush(stderr);
        exit(1);
    }

    strcpy(dce_login, argv[1]);
    strcpy(passwd, argv[2]);
    strcpy(server_principal, argv[3]);
    MAX_PASS = atoi(argv[4]);

    fprintf(stdout,"Establishing Identity with Security Server...\n");
    fflush(stdout);

    sec_login_setup_identity(dce_login, sec_login_no_flags, &login_context, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_text, &st1);
        fprintf(stderr, "Cannot set up login identity: %s\n", error_text);
        fflush(stderr);
        exit(1);
    }

    /* Check the passwd.h header file for this structure */

    pwrec.key.tagged_union.plain = passwd;
    pwrec.key.key_type = sec_passwd_plain;
    pwrec.pepper = NULL;
    pwrec.version_number = sec_passwd_c_version_none;

    ...
sec_login_validate_identity(login_context,&pwrec,&reset_passwd, &auth_src,&st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &st1);
    fprintf(stderr, "Cannot validate login identity: \%s\n", error_text);
    fflush(stderr);
    exit(1);
}

if (reset_passwd) {
    printf("Password must be changed !\n");
    fflush(stdout);
}

sec_login_set_context(login_context,&st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &st1);
    fprintf(stderr, "Cannot set login context: \%s\n", error_text);
    fflush(stderr);
    exit(1);
}

printf("Identity established !\n");
fflush(stdout);

/* Import compatible server bindings from the namespace */

printf("Importing a binding handle...\n");
fflush(stdout);

rpc_ns_binding_import_begin(rpc_c_ns_syntax_dce,
                            "/::/your_dir_name/greet6",greet_v1_0_c_ifspec,
                            NULL,&import_context,&st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &st1);
    fprintf(stderr, "Cannot begin import - \%s\n", error_text);
    exit(1);
}

rpc_ns_binding_import_next(import_context,&h,&st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &st1);
    fprintf(stderr, "Cannot import next: \%s\n", error_text);
    fflush(stderr);
    exit(1);
}

rpc_ns_binding_import_done(&import_context,&st);
if (st != error_status_ok) {
    dce_error_inq_text(st, error_text, &st1);
    fprintf(stderr, "Cannot end import - \%s\n", error_text);
    fflush(stderr);
    exit(1);
}

/* Using the "rpc_c_authz_dce" option will make the runtime send a PAC
 to the server, encrypted with the "server_principal" encryption key */

printf("Setting Auth info...\n");
fflush(stdout);
rpc_binding_set_auth_info(h, server_principal, rpc_c_protect_level_pkt_integ,
  rpc_c_authn_dce_secret, login_context, rpc_c_authz_dce, &st);
if (st != error_status_ok) {
  dce_error_inq_text(st, error_text, &st1);
  fprintf(stderr, "Cannot set auth info: %s\n", error_text);
  fflush(stderr);
  exit(1);
}
/* rpc_string_free(&server_principal, &st); */

printf("Making RPC call...\n");
fflush(stdout);

for (i=1; i <= MAX_PASS; i++) {
  greet_rpc(h, "Hello Server!", reply);
  printf("The Greet Server said: %s\n", reply);
  fflush(stdout);
}
/* Purge the login context */

printf("Purging Login context\n");
fflush(stdout);

sec_login_purge_context(&login_context, &st);
if (st != error_status_ok) {
  dce_error_inq_text(st, error_text, &st1);
  fprintf(stderr, "Cannot purge login context: %s\n", error_text);
  fflush(stderr);
  exit(1);
}
return(0);
}
Appendix D. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

**Trademarks**

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

<table>
<thead>
<tr>
<th>Brand</th>
<th>Brand</th>
<th>Brand</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX</td>
<td>BookManager</td>
<td>CICS</td>
</tr>
<tr>
<td>CICS/ESA</td>
<td>IBM</td>
<td>IBMLink</td>
</tr>
<tr>
<td>IMS</td>
<td>IMS/ESA</td>
<td>Language Environment</td>
</tr>
<tr>
<td>Library Reader</td>
<td>MVS</td>
<td>MVS/ESA</td>
</tr>
<tr>
<td>OS/390</td>
<td>RACF</td>
<td>Resource Link</td>
</tr>
<tr>
<td>SecureWay</td>
<td>VTAM</td>
<td>z/OS</td>
</tr>
</tbody>
</table>

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.
Other company, product, and service names may be trademarks or service marks of others.

Programming Interface Information

This z/OS DCE Application Development Guide: Introduction and Style documents intended Programming Interfaces that allow the customer to write programs to obtain services of z/OS DCE.
Glossary

This glossary defines technical terms and abbreviations used in z/OS DCE documentation. If you do not find the term you are looking for, refer to the index of the appropriate z/OS DCE manual or view the IBM Glossary of Computing Terms, located at:

http://www.ibm.com/ibm/terminology

This glossary includes terms and definitions from:

- Information Technology—Portable Operating System Interface (POSIX), from the POSIX series of standards for applications and user interfaces to open systems, copyrighted by the Institute of Electrical and Electronics Engineers (IEEE).
- American National Standard Dictionary for Information Systems, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.
- Information Technology Vocabulary, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1.SC1).
- Open Software Foundation (OSF).

The following abbreviations indicate terms that are related to a particular DCE service:

CDS          Cell Directory Service
CICS/ESA®    Customer Information Control System/ESA
DTS          Distributed Time Service
GDS          Global Directory Service
IMS/ESA®     Information Management System/ESA
RPC          Remote Procedure Call
Security      Security Service
Threads       Threads Service
XDS          X/Open Directory Services
XOM          X/Open OSI-Abstract-Data Manipulation

A

absolute time.  A point on a time scale.

abstract syntax notation one (ASN.1).  A data representation scheme that enables complicated types to be defined and enables values of these types to be specified.

access control list (ACL).  (1) GDS: Specifies the users with their access rights to an object.  (2) Security: Data that controls access to a protected object. An ACL specifies the privilege attributes needed to access the object and the permissions that may be granted, to the protected object, to principals that possess such privilege attributes.

access control list facility.  A Security Service feature that checks a principal's access to an object.  This facility determines access rights by comparing the principal's privileges to entries in an access control list (ACL) of an object.

access right.  Synonym for permission.

accessible.  Pertaining to an object whose client possesses a valid designator or handle.

account.  Data in the Registry database that allows a principal to log in.  An account is a registry object that relates to a principal.

ACF.  Attribute configuration file.

ACL.  Access control list.

active context handle.  RPC: A context handle in RPC applications that the RPC 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.  See client context and context handle.

address.  An unambiguous name, label, or number that identifies the location of a particular entity or service.  See presentation address.

address family.  A set of related communications protocols that use a common addressing mechanism to identify end-points; for example, the U.S. Department of Defense Internet Protocols.  Synonymous with protocol family.

alias.  Synonym for alias name.
alias entry. GDS: A directory entry, of object class alias, containing information used to provide an alternative name for an object.

alias name. (1) GDS: A name for a directory object that consists of one or more alias entries in the directory information tree (DIT). (2) Security: An optional alternate for a principal’s primary name. Synonymous with alias. The alias shares the same UUID with the primary name.

aliasing. RPC: Pertaining to the pointing of two pointers of the same operation at the same storage.

anonymous user. A user who is not entered in the directory as an object and who logs in to the Global Directory Service without giving a name and password.

APF. Authorized program facility.

API. Application program interface.

application program interface (API). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

Application Support Server. Refers to the Application Support Server/CICS and the Application Support Server/IMS.

Application Support Server/CICS. A z/OS feature. The server function in the Distributed Computing Environment (DCE) that allows a client program to access Customer Information Control System (CICS) application programs by a remote procedure call (RPC).

Application Support Server/IMS. A z/OS feature. The server function in the Distributed Computing Environment (DCE) that allows a client program to access Information Management System (IMS) application programs by a remote procedure call (RPC).

application thread. A thread of execution created and managed by application code. See client application thread, local application thread, RPC thread, and server application thread.

architecture. (1) The organizational structure of a computer system, including the interrelationships among its hardware and software. (2) The logical structure and operating principles of a computer network. The operating principles of a network include those of services, functions, and protocols.

ASN.1. Abstract syntax notation one.

association (connection-oriented). A connection between a client and a server.

asynchronous. Without a regular time relationship; unexpected or unpredictable with respect to the running of program instructions.

at-most-once semantics. RPC: A characteristic of a procedure that restricts the procedure to being run once, partially, or not at all. See broadcast semantics, idempotent semantics, and maybe semantics.

attribute. (1) RPC: An Interface Definition Language (IDL) or attribute configuration file (ACF) that conveys information about an interface, type, field, parameter, or operation. (2) DTS: A qualifier used with DTS commands. DTS has four attribute categories: characteristics, counters, identifiers, and status. (3) XDS: Information of a particular type concerning an object and appearing in an entry that describes the object in the directory information base (DIB). It denotes the attribute’s type and a sequence of one or more attribute values, each accompanied by an integer denoting the value’s syntax.

attribute configuration file (ACF). RPC: An optional companion to an interface definition file that changes how the Interface Definition Language (IDL) compiler locally interprets the interface definition. See also interface definition and Interface Definition Language.

Attribute Configuration Language. RPC: A high-level declarative language that provides syntax for attribute configuration files. See attribute configuration file.

attribute syntax. GDS: A definition of the set of values that an attribute may assume. Attribute syntax includes the data type, in ASN.1, and usually one or more matching rules by which values may be compared.

attribute type. (1) XDS: The component of an attribute that indicates the type of information given by that attribute. Because it is an object identifier, it is unique among other attribute types. (2) XOM: Any of various categories into which the client dynamically groups values on the basis of their semantics. It is an integer unique only within the package.

attribute value. XDS, XOM: A particular instance of the type of information indicated by an attribute type.

authentication. In computer security, a method used to verify the identity of a principal.

authentication level. Synonym for protection level.

authentication protocol. A formal procedure for verifying a principal’s network identity. Kerberos is an instance of a shared-secret authentication protocol.

Authentication Service. One of three services provided by the Security Service: it verifies principals.
according to a specified authentication protocol. The other Security services are the Privilege Service and the Registry Service.

**authentication surrogate.** Security: A type of principal entry in a cell’s Registry database that represents a foreign cell. This principal shares a secret key with a corresponding entry in the foreign cell’s Registry. The Authentication Services of the two cells use the secret key to exchange data about principals without either Authentication Service having to share its private key with the other.

**authorization.** (1) The determination of a principal’s permissions with respect to a protected object. (2) The approval of a permission sought by a principal with respect to a protected object.

**authorization protocol.** A formal procedure for establishing the authorization of principals with respect to protected objects. Authorization protocols supported by the Security Service include DCE authorization and name-based authorization.

**authorization service.** RPC: An implementation of an authorization protocol.

**authorized program facility (APF).** An MVS facility that permits identification of programs authorized to use restricted functions.

**automatic binding method.** RPC: A method of managing the binding for a remote procedure call. It 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. See binding handle, explicit binding method, and implicit binding method.

**B**

**big endian.** An attribute of data representation that reflects how multi-octet data is stored. In big endian representation, the lowest addressed octet of a multi-octet data item is the most significant. See little endian.

**binary timestamp.** An opaque 128-bit (16-octet) structure that represents a DTS time value.

**binding.** RPC: A relationship between a client and a server involved in a remote procedure call.

**binding handle.** RPC: A reference to a binding. See binding information.

**binding information.** RPC: Information about one or more potential bindings, including an RPC protocol sequence, a network address, an endpoint, at least one transfer syntax, and an RPC protocol version number.

See binding. See also endpoint, network address, RPC protocol, RPC protocol sequence, and transfer syntax.

**broadcast.** A notification sent to all members within an arbitrary grouping such as nodes in a network or threads in a process. See also signal.

**broadcast semantics.** RPC: A form of idempotent semantics that indicates that the operation is always broadcast to all host systems on the local network, rather than delivered to a specific system. An operation with broadcast semantics is implicitly idempotent. Broadcast semantics are supported only by connectionless protocols. See at-most-once semantics, idempotent semantics, and maybe semantics.

**Browser.** CDS: A Motif-based program that lets users view the contents and structure of a cell name space.

**C**

**C interface.** The interface that is defined at a level that depends on the variant of C standardized by ANSI.

**cache.** (1) CDS: The information that a CDS clerk stores locally to optimize name lookups. The cache contains attribute values resulting from previous lookups, as well as information about other clearinghouses and namespaces. (2) Security: Contains the credentials of a principal after the DCE login. (3) GDS: See DUA cache.

**callback.** A role reversal technique used by the server to make a request back to the original client. For example, the server may request state information (such as sequence numbers) needed to provide reliable data transfer or identity information needed for an authenticated RPC call.

**call handle.** An RPC data structure used by the RPC runtime to maintain the state information for an RPC call. A client call handle is maintained by the client, and a corresponding server call handle is maintained by the server.

**call queue.** RPC: A FIFO queue used by an RPC server to hold incoming calls when the server is already running its maximum number of concurrent calls.

**call thread.** RPC: A thread created by an RPC server’s runtime to run remote procedures. When engaged by a remote procedure call, a call thread temporarily forms part of the RPC thread of the call. See application thread and RPC thread.

**cancel.** (1) Threads: A mechanism by which a thread informs either itself or another thread to stop the thread as soon as possible. If a cancel arrives during an important operation, the canceled thread may continue until it can end the thread in a controlled manner.
CDS clerk. The software that provides an interface between client applications and CDS servers.

CDS control program (CDSCP). A command interface that CDS administrators use to control CDS servers and clerks and manage the name space and its contents. See also manager.

CDSCP. CDS control program.

cell. The basic unit of operation in the distributed computing environment. A cell is a group of users, systems, and resources that are grouped around a common purpose and that share common DCE services.

Cell Directory Service (CDS). A DCE component. A distributed replicated database service that stores names and attributes of resources located in a cell. CDS manages a database of information about the resources in a group of machines called a DCE cell.

cell-relative name. Synonym for local name.

central processing unit (CPU). The part of a computer that includes the circuits that control the interpretation and processing of instructions.

child process. A process, created by a parent process, that shares the resources of the parent process to carry out a request. Contrast with parent process. See also fork.

CICS. Customer Information Control System.

class. A category into which objects are placed on the basis of their purpose and internal structure.

clearinghouse. CDS: A collection of directory replicas on one CDS server. A clearinghouse takes the form of a database file. It can exist only on a CDS server node; it cannot exist on a node running only CDS clerk software. Usually only one clearinghouse exists on a server node.

clerk. (1) DTS: A software component that synchronizes the clock for its client system by requesting time values from servers, calculating a new time from the values, and supplying the computed time to client applications. (2) CDS: A software component that receives CDS requests from a client application, ascertains an appropriate CDS server to process the requests, and returns the results of the requests to the client application.

client. A computer or process that accesses the data, services, or resources of another computer or process on the network. Contrast with server.

client application thread. RPC: A thread executing client application code that makes one or more remote procedure calls. See application thread, local application thread, RPC thread, and server application thread.

client binding information. 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 client authentication information. See binding information and server binding information.

client context. RPC: The state within an RPC server generated by a set of remote procedures and maintained across a series of calls for a particular client. See context handle. See also manager.

client stub. RPC: 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, to manage bindings. See server stub.

client/server model. A form of computing where one system, the client, requests something, and another system, the server, responds.

clock. The combined hardware interrupt timer and software register that maintains the system time.

code page. (1) A table showing codes assigned to character sets. (2) An assignment of graphic characters and control function meanings to all code points. (3) Arrays of code points representing characters that establish numeric order of characters. [OSF] (4) A particular assignment of hexadecimal identifiers to graphic elements. (5) Synonymous with code set. (6) See also code point, extended character.

code set. Synonym for code page.

collapse. CDS: To remove the contents of a directory from the display (close it) using the CDS Browser. To collapse an open directory, double-click on its icon. Double-clicking on a closed directory expands it. Contrast with expand.

communications link. RPC: A network pathway 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 run times.
compatible server. RPC: A server that offers the requested RPC interface and RPC object and that is accessible over a valid combination of network and transport protocols. It is supported by both the client and server RPC run times.

computed time. DTS: The resulting time after a DTS clock synchronization. The time value that the clerk or server process computes according to the values it receives from several servers.

condition variable. Threads: A synchronization object used in conjunction with a mutex. It allows a thread to suspend running until some condition is true.

conformant array. RPC: An array whose size is determined at runtime. A structure containing a conformant array as a field is a conformant structure.

connectionless protocol. RPC: A transport protocol such as UDP that does not require a connection to be established prior to data transfer. Contrast with connection-oriented protocol.

connection-oriented protocol. RPC: An RPC protocol that runs over a connection-based transport protocol. It is a connection-based, reliable, virtual-circuit transport protocol, such as TCP. Contrast with connectionless protocol.

Consultative Committee on International Telegraphy and Telephone (CCITT). A United Nations Specialized Standards group whose membership includes common carriers concerned with devising and proposing recommendations for international telecommunications representing alphabets, graphics, control information, and other fundamental information interchange issues.

context handle. RPC: A reference to state (client context) maintained across remote procedure calls by a server on behalf of a client. See client context.

customized binding handle. RPC: A user-defined data structure from which a primitive binding handle can be derived by user-defined routines in application code. See primitive binding handle.

daedron. (1) A long-lived process that runs unattended to perform continuous or periodic system-wide functions such as network control. Some daemons are triggered automatically to perform their task; others operate periodically. An example is the cron daemon, which periodically performs the tasks listed in the crontab file. Many standard dictionaries accept the spelling demon. (2) A DCE server process.

as Federal Information Processing Standard (FIPS) Publication 46, which allows only hardware implementations of the data encryption algorithm.

data limit.  RPC: A value that specifies which elements of an array are transmitted during a remote procedure call.

datagram.  RPC: A network data packet that is independent of all other packets and does not guarantee delivery or sequentiality.

datagram protocol.  RPC: A datagram-based transport protocol, such as User Datagram Protocol (UDP), that runs over a connectionless transport protocol.

DBCS.  Double-byte character sets

DCE.  Distributed Computing Environment.

DCEKERN.  The address space that contains the DCE daemons.

decrypt.  Security: To decipher data.

default element.  RPC: An optional profile element that contains a nil interface identifier and object UUID and that specifies a default profile. Each profile can contain only one default element. See default profile, profile, and profile element.

default profile.  RPC: A backup profile referred to by the default element in another profile. The NSI import and lookup operations use the default profile, if present, whenever a search based on the current profile fails to find any useful binding information. See default element and profile.

DES.  Data Encryption Standard.

descr iptor.  (1) XOM: The means by which the client and service exchange an attribute value and the integers that denote its representation, type, and syntax. (2) XDS: A defined data structure that is used to represent an OM attribute type and a single value.

destructor.  A user-supplied routine that is expected to finalize and then deallocate a per-thread context value.

DFS.  Distributed File Service.

DIB.  Directory information base.

directory.  (1) A logical unit for storing entries under one name (the directory name) in a CDS namespace. Each physical instance of a directory is called a replica. (2) A collection of open systems that cooperates to hold a logical database of information about a set of objects in the real world.

directory information base (DIB).  GDS: The complete set of information to which the directory provides access, which includes all of the pieces of information that can be read or manipulated using the operations of the directory.

directory information tree (DIT).  GDS: The directory information base (DIB) considered as a tree, whose vertices (other than the root) are the directory entries.

directory schema.  GDS: The set of rules and constraints concerning directory information tree (DIT) structure, object class definitions, attribute types, and syntaxes that characterize the directory information base (DIB).


directory system.  GDS: A system for managing a directory, consisting of one or more DSAs. Each DSA manages part of the DIB.

directory system agent (DSA).  GDS: An open systems interconnection (OSI) application process that is part of the directory.

directory system protocol (DSP).  GDS: The protocol used by a directory system agent (DSA) to access another DSA. The DSA runs in the GDS server machine and manages the GDS data base.

directory user agent (DUA).  GDS: An open systems interconnection (OSI) application process that represents a user accessing the directory.

discrimin ator.  RPC: The data item that determines which union case is currently used.

distributed computing.  A type of computing that allows computers with different hardware and software to be combined on a network, to function as a single computer, and to share the task of processing application programs.

Distributed Computing Environment (DCE).  A comprehensive, integrated set of services that supports the development, use, and maintenance of distributed applications. DCE is independent of the operating system and network; it provides interoperability and portability across heterogeneous platforms.

Distributed File Service (DFS).  A DCE component. DFS joins the local file systems of several file server machines making the files equally available to all DFS client machines. DFS allows users to access and share files stored on a file server anywhere in the network, without having to consider the physical location of the file. Files are part of a single, global name space, so
that a user can be found anywhere in the network by means of the same name.

**distributed service.** A DCE service that is used mainly by administrators to manage a distributed environment. These services include DTS, Security, and Directory.

**Distributed Time Service (DTS).** A DCE component. It provides a way to synchronize the times on different hosts in a distributed system.

**DIT.** Directory information tree.

**DNS.** Domain Name System.

**Domain Name System (DNS).** A hierarchical scheme for giving meaningful names to hosts in a TCP/IP network.

**Double-byte character sets (DBCS).** A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte coded character sets.

**domain name.** A unique network name that is associated with a network's unique address.

**DSA.** Directory system agent.

**DSP.** Directory system protocol.

**DTS.** Distributed Time Service.

**DTS entity.** DTS: The server or clerk software on a system.

**DUA.** Directory user agent.

**DUA cache.** GDS: The part of the DUA that stores information to optimize name lookups. Each cache contains copies of recently accessed object entries as well as information about DSAs in the directory.

**dynamic endpoint.** RPC: An endpoint that is generated by the RPC runtime for an RPC server when the server registers its protocol sequences. It expires when the server stops running. See endpoint and well-known endpoint.

**E**

**effective permissions.** Security: The permissions granted to a principal as a result of a masking operation.

**element.** RPC: Any of the bits of a bit string, the octets of an octet string, or the octets by means of which the characters of a character string are represented.

**encrypt.** To systematically encode data so that it cannot be read without knowing the coding key.

**encryption key.** A value used to encrypt data so that only possessors of the encryption key can decipher it.

**endian.** An attribute of data representation that reflects how certain multi-octet data is stored in memory. See big endian and little endian.

**endpoint.** RPC: An address of a specific server instance on a host.

**endpoint map.** RPC: A database local to a node where local RPC servers register binding information associated with their interface identifiers and object identifiers. The endpoint map is maintained by the endpoint map service of the DCE daemon.

**endpoint map service.** RPC: A service that maintains a system's endpoint map for local RPC servers. When an RPC client makes a remote procedure call using a partially bound binding handle, the endpoint map service looks up the endpoint of a compatible local server. See endpoint map.

**entity.** (1) CDS: Any manageable element through the CDS namespace. Manageable elements include directories, object entries, servers, replicas, and clerks. The CDS control program (CDSCP) commands are based on directives targeted for specific entities. (2) DTS: See DTS entity.

**entry.** GDS, XDS: The part of the DIB that contains information relating to a single directory object. Each entry consists of directory attributes.

**entry point vector (EPV).** RPC: A list of addresses for the entry points of a set of remote procedures that starts the operations declared in an interface definition. The addresses are listed in the same order as the corresponding operation declarations.

**ENV.** environment variable

**envelope.** Security: Used to transport authentication data and conversation keys between the security server and principals.

**environment variable (ENV).** A variable included in the current software environment that is available to any called program that requests it.

**EPV.** Entry point vector.

**exception.** (1) An abnormal condition such as an I/O error encountered in processing a data set or a file. (2) One of five types of errors that can occur during a
floating-point exception. These are valid operation, overflow, underflow, division by zero, and inexact results. [OSF] (3) Contrast with interrupt, signal.

executor thread. See call thread.

expand. CDS: To display the contents of (open) a directory using the CDS Browser. A directory that is closed can be expanded by double-clicking on its icon. Double-clicking on an expanded directory collapses it. Contrast with collapse.

expiration age. RPC: The amount of time that a local copy of name service data from a NSI attribute remains unchanged before a request from an RPC application for the attribute requires its updating. See also NSI attribute.

explicit binding method. RPC: 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 is initialized in the application code. See automatic binding method, binding handle, and implicit binding method.

export. (1) RPC: To place the server binding information associated with an RPC interface or a list of object UUIDs or both into an entry in a name service database. (2) To provide access information for an RPC interface. Contrast with unexport.

F

fault. RPC: An exception condition, occurring on a server, that is transmitted to a client.

filter. An assertion about the presence or value of certain attributes of an entry to limit the scope of a search.

FIFO. first-in-first-out

first-in-first-out (FIFO). A queueing technique in which the next item to be retrieved is the item that has been in the queue the longest time.

fixed array. RPC: The size of the array is defined in the IDL. All of the data in the array is transmitted during a remote procedure call.

foreign cell. A cell other than the one to which the local machine belongs. A foreign cell and its binding information are stored in either GDS or the Domain Name System (DNS). The act of contacting a foreign cell is called intercell. Contrast with local cell.

fork. To create and start a child process. Forking is similar to creating an address space and attaching. It creates a copy of the parent process, including open file descriptors.

full name. CDS: The complete specification of a CDS name, including all parent directories in the path from the cell root to the entry being named.

full pointer. RPC: A pointer without the restrictions of a reference pointer.

fully bound binding handle. RPC: A server binding handle that contains a complete server address including an endpoint. Contrast with partially bound binding handle.

G

General-Use Programming Interface (GUPI). An interface, with few restrictions, for use in customer-written programs. The majority of programming interfaces are general-use programming interfaces, and are appropriate in a wide variety of application programs. A general-use programming interface requires the knowledge of the externals of the interface and perhaps the externals of related programming interfaces. Knowledge of the detailed design or implementation of the software product is not required.

GDA. Global Directory Agent.

GDS. Global Directory Service.

Global Directory Agent (GDA). A DCE component that makes it possible for the local CDS to access names in foreign cells. The GDA provides a connection to foreign cells through either the GDS or the Domain Name System (DNS).

Global Directory Service (GDS). A DCE component. A distributed replicated directory service that provides a global namespace that connects the local DCE cells into one worldwide hierarchy. DCE users can look up a name outside a local cell with GDS.

global name. A name that is universally meaningful and usable from anywhere in the DCE naming environment. The prefix /... indicates that a name is global.

global server. DTS: A server that provides its clock value to courier servers on other cells, or to DTS entities that have failed to obtain the specified number of servers locally.

group. (1) RPC: A name service entry that corresponds to one or more RPC servers that offer common RPC interfaces, RPC objects, or both. A group contains the names of the server entries, other groups, or both that are members of the group. See NSI group attribute. (2) Security: Data that associates
a named set of principals that can be granted common access rights. See subject identifier.

**group member.** (1) RPC: A name service entry whose name occurs in the group. (2) Security: A principal whose name appears in a security group. See group.

**GUPI.** General-Use Programming Interface.

**H**

**handle.** RPC: An opaque reference to information. See binding handle, context handle, interface handle, name service handle, and thread handle.

**heterogeneous.** Pertaining to a collection of dissimilar host computers such as those from different manufacturers. Contrast with homogeneous.

**home cell.** Synonym for local cell.

**homogeneous.** Pertaining to a collection of similar host computers such as those of one model or one manufacturer. Contrast with heterogeneous.

**host ID.** Synonym for network address.

**I**

**idempotent semantics.** RPC: A characteristic of a procedure in which running it more than once with identical input always produces the same result, without any undesirable side effects. For example, a procedure that calculates the square root of a number is idempotent. DCE RPC supports maybe and broadcast semantics as special forms of idempotent operations. See at-most-once semantics, broadcast semantics, and maybe semantics.

**IDL.** Interface Definition Language.

**IDL compiler.** RPC: A compiler that processes an RPC interface definition and an optional attribute configuration file (ACF) to generate client and server stubs, and header files. See Interface Definition Language.

**implicit binding method.** RPC: The implicit method of managing the binding for a remote procedure call in which a global variable in the client application holds a binding handle that the client stub passes to the RPC runtime. See automatic binding method, binding handle, and explicit binding method.

**import.** (1) RPC: To obtain binding information from a name service database about a server that offers a given RPC interface by calling the RPC NSI import operation. (2) RPC: To incorporate constant, type, and import declarations from one RPC interface definition into another RPC interface definition by means of the IDL import statement.

**import context.** The context set up by the client to import compatible binding handles from the name space. Name service interfaces (NSI) are used to set up and free the import context.

**IMS.** Information Management System.

**inaccuracy.** DTS: The bounded uncertainty of a clock value as compared to a standard reference.

**Information Management System (IMS).** A database and data communication system capable of managing complex databases and networks in virtual storage.

**inquiry context.** The context set up by the client, server, or management applications to view the elements in a name space profile. Name service interfaces (NSI) are used to set up and free the inquiry context.

**instance.** XOM: An object in the category represented by a class.

**instance UUID.** RPC: An object Universal Unique Identifier (UUID) that is associated with a single server instance and is provided to clients to identify that instance unambiguously. See object UUID and server instance.

**integrity.** RPC: A protection level that may be specified in secure RPC communications to ensure that data transferred between two principals has not been changed in transit.

**interface.** RPC: A shared boundary between two or more functional units, defined by functional characteristics, signal characteristics, or other characteristics, as appropriate. The concept includes the specification of the connection of two devices having different functions. See RPC interface.

**interface definition.** RPC: A description of an RPC interface written in the DCE Interface Definition Language (IDL). See RPC interface.

**Interface Definition Language (IDL).** A high-level declarative language that provides syntax for interface definitions.

**interface handle.** RPC: A reference in code to an interface specification. See binding handle and interface specification.

**interface identifier.** RPC: A string containing the interface Universal Unique Identifier (UUID) and major and minor version numbers of a given RPC interface. See RPC interface.
interface specification.  RPC: An opaque data structure that is generated by the DCE IDL compiler from an interface definition. It contains identifying and descriptive information about an RPC interface. See interface definition, interface handle, and RPC interface.

interface UUID.  RPC: The Universal Unique Identifier (UUID) generated for an RPC interface definition using the UUID generator. See interface definition and RPC interface.

International Organization for Standardization (ISO).  An international body composed of the national standards organizations of 89 countries. ISO issues standards on a vast number of goods and services including networking software.

Internet address.  The 32-bit address assigned to hosts in a TCP/IP network.

Internet Protocol (IP).  In TCP/IP, a protocol that routes data from its source to its destination in an Internet environment. IP provides the interface from the higher level host-to-host protocols to the local network protocols. Addressing at this level is usually from host to host.

interoperability.  The capability to communicate, execute programs, or transfer data among various functional units in a way that requires the user to have little or no knowledge of the unique characteristics of those units.

interval.  DTS: The combination of a time value and the inaccuracy associated with it; the range of values represented by a combined time and inaccuracy notation. As an example, the interval 08:00.00100:05:00 (eight o’clock, plus or minus five minutes) contains the time 07:57:00.

IP.  Internet Protocol

ISO.  International Organization for Standardization

J

junction.  A specialized entry in the DCE namespace that contains binding information to enable communications between different DCE services.

K

Kerberos.  The authentication protocol used to carry out DCE private key authentication. Kerberos was developed at the Massachusetts Institute of Technology.

key.  A value used to encrypt and decrypt data.

key file.  A file that contains encryption keys for noninteractive principals.

key management facility.  A Security Service facility that enables noninteractive principals to manage their secret keys.

L

LAN.  Local area network.

layer.  In network architecture, a group of services, functions, and protocols that is complete from a conceptual point of view, that is one out of a set of hierarchically arranged groups, and that extends across all systems that conform to the network architecture.

leaf entry.  A directory entry that has no subordinates. It can be an alias entry or an object entry.

LFS.  local file system

listener thread.  Created by RPC to listen on all TCP/IP sockets for calls coming into the client for the datagram protocol and for calls coming into the server for datagram and connection-oriented protocols.

little endian.  An attribute of data representation that reflects how multi-octet data is stored. In little endian representation, the lowest addressed octet of a multi-octet data item is the least significant. See big endian.

liveness.  Context handle and related maintenance functions that maintain context on behalf of clients even during periods of nominal client inactivity.

local.  (1) Pertaining to a device directly connected to a system without the use of a communication line.  (2) Pertaining to devices that have a direct, physical connection. Contrast with remote.

local application thread.  RPC: An application thread that runs within the confines of one address space on a local system and passes control exclusively among local code segments. See application thread, client application thread, RPC thread and server application thread.

local area network (LAN).  A network in which communication is limited to a moderate-sized geographical area (1 to 10 km) such as a single office building, warehouse, or campus, and which does not generally extend across public rights-of-way. A local network depends on a communication medium capable of moderate to high data rate (greater than 1Mbps), and normally operates with a consistently low error rate.
**local cell.** The cell to which the local machine belongs. Synonymous with *home cell.* Contrast with *foreign cell.*

**local file system (LFS).** An organized collection of data in the form of a root directory and its subdirectories and files. An LFS supports special features useful in a distributed environment: the ability to replicate data; to log file system data, enabling quick recovery after a crash; to simplify administration by dividing the file system into easily managed units called filesets; and to associate access control lists (ACLs) with files and directories. An LFS is located on a disk that is physically attached to a machine. In other file systems, a single disk partition contains only one file system. In DCE, LFS can contain multiple file systems (filesets). See also *access control list (ACL).*

**local name.** A name that is meaningful and usable only within the cell where an entry exists. The local name is a shortened form of a global name. Local names begin with the prefix `/.:` and do not contain a cell name. Synonymous with *cell-relative name.*

**local server.** DTS: A server that synchronizes with its peers and provides its clock value to other servers and clerks in the same network.

**local type.** RPC: A type named in a *represent_as* clause and used by application code to manipulate data that is passed in a remote procedure call as a network type. See *network type.*

**logical unit (LU).** A host port through which a user gains access to the services of a network.

**login facility.** A Security Service facility that enables a principal to establish its identity.

**LU.** Logical unit.

**M**

**manager.** RPC: A set of remote procedures that implement the operations of an RPC interface and that can be dedicated to a given type of object. See also *object and RPC interface.*

**manager entry point vector.** RPC: The runtime code on the server side uses this entry point vector to dispatch incoming remote procedure calls. See *entry point vector* and *manager.*

**manager thread.** See *call thread.*

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

**mask.** (1) A pattern of characters used to control the retention or deletion of portions of another pattern of characters (2) Security: Used to establish maximum permissions that can then be applied to individual ACL entries. (3) GDS: The administration screen interface menus.

**master replica.** CDS: The first instance of a specific directory in the namespace. After copies of the directory have been made, a different replica can be designated as the master, but only one master replica of a directory can exist at a time. CDS can create, update, and delete object entries and soft links in a master replica.

**maybe semantics.** RPC: A form of idempotent semantics that indicates that the caller neither requires nor receives any response or fault indication for an operation, even though there is no guarantee that the operation was completed. An operation with maybe semantics is implicitly idempotent and lacks output parameters. See *at-most-once semantics, broadcast semantics,* and *idempotent semantics.*

**mutex.** Mutual exclusion. A read/write lock that grants access to only a single thread at any one time. A mutex is often used to ensure that shared variables are always seen by other threads in a consistent way.

**mvsexpt.** One of two (the other is *mvsimpt*) utilities used to automate much of the administrator’s work in creating the cross-linking information for DCE-RACF interoperability. The *mvsexpt* utility creates the cross-linking information in the RACF database from information in the DCE registry. See also *cross-linking information, interoperability,* and *single sign-on.*

**mvsimpt.** One of two (the other is *mvsexpt*) utilities used to automate much of the administrator’s work in creating the cross-linking information for DCE-RACF interoperability. The *mvsimpt* utility creates DCE principals from information obtained from the RACF database. See also *cross-linking information, interoperability,* and *single sign-on.*

**N**

**name.** GDS, CDS: A construct that singles out a particular (directory) object from all other objects. A name must be unambiguous (denote only one object); however, it need not be unique (be the only name that unambiguously denotes the object).

**name service.** A central repository of named resources in a distributed system. In DCE, this is the same as Directory Service.

**name service handle.** RPC: An opaque reference to the context used by the series of next operations called...
during a specific name service interface (NSI) search or inquiry.

**name service interface (NSI).** RPC: A part of the application program interface (API) of the RPC run time. NSI routines access a name service, such as CDS, for RPC applications.

**namespace.** CDS: A complete set of CDS names that one or more CDS servers look up, manage, and share. These names can include directories, object entries, and soft links.

**NCA.** Network Computing Architecture.

**NDR.** Network Data Representation.

**network.** A collection of data processing products connected by communications lines for exchanging information between stations.

**network address.** An address that identifies a specific host on a network. Synonymous with *host ID*.

**Network Computing Architecture (NCA).** RPC: An architecture for distributing software applications across heterogeneous collections of networks, computers, and programming environments using UDP. NCA specifies part of the DCE Remote Procedure Call architecture.

**network data.** RPC: Data represented in a format defined by a transfer syntax. See also *transfer syntax*.

**Network Data Representation (NDR).** RPC: The transfer syntax defined by the Network Computing Architecture. See *transfer syntax*.

**network descriptor.** RPC: The identifier of a potential network channel, such as a UNIX socket.

**network protocol.** A communications protocol from the Network Layer of the Open Systems Interconnection (OSI) network architecture, such as the Internet Protocol (IP).

**Network Time Protocol (NTP).** A clock synchronization protocol commonly used on an Internet.

**network type.** RPC: A type defined in an interface definition and referenced in a *represent as* clause that is converted into a local type for manipulation by application code. See *local type*.

**node.** (1) An endpoint of a link, or a junction common to two or more links in a network. Nodes can be preprocessors, controllers, or workstations, and they can vary in routing and other functional capabilities. (2) In network topology, the point at an end of a branch. It is usually a physical machine.

**non-idempotent.** An RPC call attribute type describing an RPC call that must run no more than once. Before running a non-idempotent call, servers and clients verify each other’s identity using one of the simple conversation callback operations provided by a set of conversation manager routines for the datagram RPC protocol service.

**null time provider.** The daemon that fetches the time from the hardware clock of the DCE host for DTS.

**NSI.** Name service interface.

**NSI attribute.** RPC: An RPC-defined attribute of a name service entry used by the RPC name service interface. A name service interface (NSI) attribute stores one of the following: binding information, object Universal Unique Identifiers (UUIDs), a group, or a profile. See *NSI binding attribute*, *NSI group attribute*, *NSI object attribute*, and *NSI profile attribute*.

**NSI binding attribute.** RPC: An RPC-defined attribute (NSI attribute) of a name service entry; the binding attribute stores binding information for one or more interface identifiers offered by an RPC server and identifies the entry as an RPC server entry. See *binding information* and *NSI object attribute*. See also *server entry*.

**NSI group attribute.** RPC: An RPC-defined attribute (NSI attribute) of a name service entry that stores the entry names of the members of an RPC group and identifies the entry as an RPC group. See *group*.

**NSI object attribute.** RPC: An RPC-defined attribute (NSI attribute) of a name service entry that stores the object UUIDs of a set of RPC objects. See *object*.

**NSI profile attribute.** RPC: An RPC-defined attribute (NSI attribute) of a name service entry that stores a collection of RPC profile elements and identifies the entry as an RPC profile. See *profile*.

**NTP.** Network Time Protocol.

**NULL.** In the C language, a pointer that does not point to a data object.

**O**

**object.** (1) A data structure that implements some feature and has an associated set of operations. (2) RPC: For RPC applications, anything that an RPC server defines and identifies to its clients using an object Universal Unique Identifier (UUID). An RPC object is often 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, such as a service or the location of a server. See *object UUID*. (3) XDS: Anything in the
world of telecommunications and information processing that can be named and for which the directory information base (DIB) contains information. (4) XOM: Any of the complex information objects created, examined, changed, or destroyed by means of the interface.

**object class.** GDS, CDS: An identified family of objects that share certain characteristics. An object class can be specific to one application or shared among a group of applications. An application interprets and uses an entry’s class-specific attributes based on the class of the object that the entry describes.

**object class table (OCT).** A recurring attribute of the directory schema with the description of the object classes permitted.

**object entry.** CDS: The name of a resource (such as a node, disk, or application) and its associated attributes, as stored by CDS. CDS administrators, client application users, or the client applications themselves can give a resource an object name. CDS supplies some attribute information (such as a creation timestamp) to become part of the object, and the client application may supply more information for CDS to store as other attributes. See *entry*.

**object identifier (OID).** A value (distinguishable from all other such values) that is associated with an information object. It is formally defined in the CCITT X.208 standard.

**object management (OM).** The creation, examination, change, and deletion of potentially complex information objects.

**object name.** CDS: A name for a network resource.

**object UUID.** RPC: The 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 stores the object. See *object*.

**octal.** In reference to a selection, choice or condition that has eight possible different values or states. In reference to a fixed-radix numeration having a radix of eight.

**OCT.** Object class table.

**octet.** A byte that consists of eight bits.

**OID.** Object identifier.

**OM.** Object management.

**OM attribute.** XOM: An object management (OM) attribute consists of one or more values of a particular type (and therefore syntax).

**opaque.** A datum or data type whose contents are not visible to the application routines that use it.

**Open Software Foundation (OSF).** A nonprofit research and development organization set up to encourage the development of solutions that allow computers from different vendors to work together in a true open-system computing environment.

**open system.** A system whose characteristics comply with standards made available throughout the industry and that can be connected to other systems complying with the same standards.

**open systems interconnection (OSI).** The interconnection of open systems in accordance with standards of the International Organization for Standardization (ISO) for the exchange of information.

**operation.** (1) GDS: Processing performed within the directory to provide a service, such as a read operation. (2) RPC: The task performed by a routine or procedure that is requested by a remote procedure call.

**organization.** (1) The third field of a subject identifier. (2) Security: Data that associates a named set of users who can be granted common access rights that are usually associated with administrative policy.

**orphaned call.** RPC: A call running in an RPC server after the client that started the call fails or loses communication with the server.

**OSF.** Open Software Foundation.

**OSI.** Open systems interconnection

**P**

**PAC.** Privilege attribute certificate.

**package.** XOM: A specified group of related object management (OM) classes, denoted by an object identifier.

**packet.** (1) In data communication, a sequence of binary digits, including data and control signals, that is transmitted and switched as a composite whole. [1] The data, call control signals, and error control information are arranged in a specific format. (2) See *call-accepted packet, call-connected packet, call-request packet, clear-confirmation packet, clear-indication packet, clear-request packet*. See *data packet, incoming-call packet*.
parent directory.  CDS: Any directory that has one or more levels of directories beneath it in a cell name space.  A directory is the parent of any directory immediately beneath it in the hierarchy.

parent process.  A process created to carry out a program.  The parent process in turn creates child processes to process requests.  Contrast with child process.

partially bound binding handle.  RPC: A server binding handle that contains an incomplete server address lacking an endpoint.  Contrast with fully bound binding handle.

Partitioned data set (PDS).  A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.

password.  A secret string of characters shared between a computer system and a user.  The user must specify the character string to gain access to the system.

PCS.  Portable Character Set.

PDS.  Partitioned data set

peer trust.  A type of trust relationship established between two cells by means of a secret key shared by authentication surrogates maintained by the two cells.  A peer trust relationship enables principals in one cell to communicate securely with principals in the other.

permission.  (1) The modes of access to a protected object.  The number and meaning of permissions with respect to an object are defined by the access control list (ACL) Manager of the object. (2) GDS: One of five groups that assigns modes of access to users: MODIFY PUBLIC, READ STANDARD, MODIFY STANDARD, READ SENSITIVE, or MODIFY SENSITIVE.  Synonymous with access right.  See also access control list.

dependent name.  The string name of an object to which any aliases for that object refer.  The DCE refers to

pipe.  (1) RPC: A mechanism for passing large amounts of data in a remote procedure call.  (2) The data structure that represents this mechanism.

plaintext.  The input to an encryption function or the output of a decryption function.  Encryption transforms plaintext to ciphertext and decryption transforms ciphertext into plaintext.

platform.  The operating system environment in which a program runs.

port.  (1) Part of an Internet Protocol (IP) address specifying an endpoint.  (2) To make the programming changes necessary to allow a program that runs on one type of computer to run on another type of computer.

Portable Character Set.  A set of characters to enable internationalization.  A character set used by DCE to enable word wide connectivity by ensuring that a minimum group of characters is supported in DCE.  All DCE RPC clients and servers are required to support the DCE PCS.

position (within a string).  XOM: The ordinal position of one element of a string relative to another.

position (within an attribute).  XOM: The ordinal position of one value relative to another.

potential binding.  RPC: A specific combination of an RPC protocol sequence, RPC protocol major version, network address, endpoint, and transfer syntax that an RPC client can use to establish a binding with an RPC server.  See binding.  See also endpoint, network address, RPC protocol, RPC protocol sequence, and transfer syntax.

predicate.  A Boolean logic term denoting a logical expression that determines the state of some variables.  For example, a predicate can be an expression stating that variable A must have the value 3.  The control expression used in conjunction with condition variables is based on a predicate.  A condition variable can be used to wait for some predicate to become true, for example, to wait for something to be in a queue.

presentation address.  An unambiguous name that is used to identify a set of presentation service access points.  Loosely, it is the network address of an open systems interconnection (OSI) service.

presented type.  RPC: For data types with the Interface Definition Language (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.  See transmitted type.

primary name.  The string name of an object to which any aliases for that object refer.  The DCE refers to
objects by their primary names, although DCE users may refer to them by their aliases.

**primitive binding handle.** RPC: A binding handle whose data type in Interface Definition Language (IDL) is `handle_t` and in application code is `rpc_binding_handle_t`. See *customized binding handle*.

**principal.** Security: An entity that can communicate securely with another entity. In the DCE, principals are represented as entries in the Registry database and include users, servers, computers, and authentication surrogates.

**privacy.** RPC: A protection level that encrypts RPC argument values in secure RPC communications.

**private key.** See *secret key*.

**privilege attribute.** Security: An attribute of a principal that may be associated with a set of permissions. DCE privilege attributes are identity-based and include the principal's name, group memberships, and local cell.

**privilege attribute certificate (PAC).** Security: Data describing a principal's privilege attributes that has been certified by an authority. In the DCE, the Privilege Service is the certifying authority; it seals the privilege attribute data in a ticket. The authorization protocol, DCE Authorization, determines the permissions granted to principals by comparing the privilege attributes in PACs with entries in an access control list.

**privilege service.** Security: One of three services provided by the Security Service: the Privilege Service certifies a principal's privileges. The other services are the Registry Service and the Authentication Service.

**privilege ticket.** Security: A ticket that contains the same information as a simple ticket, and also includes a privilege attribute certificate. See *service ticket*, *simple ticket*, and *ticket-granting ticket*.

**procedure declaration.** RPC: The syntax for an operation, including its name, the data type of the value it returns (if any), and the number, order, and data types of its parameters (if any).

**product-sensitive programming interface (PSPI).** (1) A special interface that is intended only to be used for a specialized task such as diagnosis, modification, monitoring, repairing, tailoring, or tuning. (2) A special interface that is dependent on or requires the customer to understand significant aspects of the detailed design and implementation of the IBM software product.

**profile.** RPC: An entry in a name service database that contains a collection of elements from which name service interface (NSI) search operations construct search paths for the database. Each search path is composed of one or more elements that refer to name service entries corresponding to a given RPC interface and, optionally, to an object. See *NSI profile attribute and profile element*.

**profile element.** RPC: A record in an RPC profile that maps an RPC interface identifier to a profile member (a server entry, group, or profile in a name service database). See *profile*. See also *group, interface identifier and server entry*.

**profile member.** RPC: A name service entry whose name occupies the member field of an element of the profile. See *profile*.

**programming interface.** The supported method through which customer programs request software services. The programming interface consists of a set of callable services provided with the product.

**proprietary.** Pertaining to the holding of the exclusive legal rights in making, using, or marketing a product.

**protection level.** The degree to which secure network communications are protected. Synonymous with *authentication level*.

**protocol.** A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication.

**protocol family.** Synonym for *address family*.

**protocol sequence.** Synonym for *RPC protocol sequence*.

**protocol sequence vector.** RPC: A data structure that contains an array-size count and an array of pointers to RPC protocol-sequence strings. See *RPC protocol sequence*.

**PSPI.** product-sensitive programming interface

**Q**

**quiescent state.** Application Support Server: The server state wherein the server can process only management calls. The Application Support Server enters this state when the server initialization has been completed or when the server is stopped by an administrative client.

**R**

**RACF.** Resource Access Control Facility.

**read access.** CDS: An access right that grants the ability to view data.

**read-only replica.** (1) CDS: A copy of a CDS
directory in which applications cannot make changes. Although applications can look up information (read) from it, they cannot create, change, or delete entries in a read-only replica. Read-only replicas become consistent with other, changeable replicas of the same directory during skulks and routine propagation of updates. (2) Security: A replicated Registry server.

**realm.** Security: A cell, considered exclusively from the point of view of Security; this term is used in Kerberos specifications. The term cell designates the basic unit of DCE configuration and administration and incorporates the notion of a realm.

**recurring attribute.** An attribute with several attribute values.

**reference monitor.** Code that controls access to an object. In the DCE, servers control access to the objects they maintain; for a given object, the ACL manager associated with that object makes authorization decisions concerning the object.

**reference pointer.** RPC: A non-null pointer whose value is invariant during a remote procedure call and cannot point at aliased storage.

**referral.** GDS: An outcome that can be returned by a DSA that cannot perform an operation itself. The referral identifies one or more other DSAs more able to perform the operation.

**register.** (1) RPC: To list an RPC interface with the RPC runtime. (2) To place server-addressing information into the local endpoint map. (3) To insert authorization and authentication information into binding information. See endpoint map and RPC interface.

**Registry database.** Security: A database of security information about principals, groups, organizations, accounts, and security policies.

**Registry Service.** Security: One of three services provided by the Security Service; the Registry Service manages information about principals, accounts, and security policies. The other services are the Privilege Service and the Authentication Service.

**relative time.** A discrete time interval that is usually added to or subtracted from an absolute time. See absolute time.

**remote.** Pertaining to a device, file or system that is accessed by your system through a communications line. Contrast with local.

**remote procedure.** RPC: An application procedure located in a separate address space from calling code. See remote procedure call.

**remote procedure call.** RPC: A client request to a service provider located anywhere in the network.

**Remote Procedure Call (RPC).** A DCE component. It allows requests from a client program to access a procedure located anywhere in the network.

**replica.** CDS: A directory in the CDS namespace. The first instance of a directory in the name space is the master replica. See master replica and read-only replica.

**replication.** The making of a shadow of a database to be used by another node. Replication can improve availability and load-sharing.

**request.** A command sent to a server over a connection.

**request buffer.** RPC: A FIFO queue where an RPC system temporarily stores call requests that arrive at an endpoint of an RPC server until the server can process them.

**resource.** Items such as printers, plotters, data storage, or computer services. Each has a unique identifier associated with it for naming purposes.

**Resource Access Control Facility (RACF).** An IBM licensed program, that provides for access control by identifying and verifying the users to the system, authorizing access to protected resources, and logging the detected unauthorized access to protected resources.

**return value.** A function result that is returned in addition to the values of any output or input/output arguments.

**ROM.** Read-only memory.

**RPC.** Remote Procedure Call.

**RPC control program (RPCCP).** An interactive administrative facility for managing name service entries and endpoint maps for RPC applications.

**RPCCP.** RPC control program

**RPC interface.** A logical group of operations, data types, and constant declarations that serves as a network contract for a client to request a procedure in a server. See also interface definition and operation.

**RPC protocol.** An RPC-specific communications protocol that supports the semantics of the DCE RPC API and runs over either connectionless or connection-oriented communications protocols.

**RPC protocol sequence.** A valid combination of communications protocols represented by a character
string. Each RPC protocol sequence typically includes three protocols: a network protocol, a transport protocol, and an RPC protocol that works with the network and transport protocols. See network protocol, RPC protocol, and transfer protocol. Synonymous with protocol sequence.

**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. See **RPC runtime library.**

**RPC runtime library.** A group of routines of the RPC runtime that support the RPC applications 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). See **RPC runtime.**

**RPC thread.** A logical thread within which a remote procedure call is executed. See **thread.**

**rundown procedure.** RPC: A procedure used with a context handle that is called following a communications failure. It recovers resources reserved by a server for servicing requests by a particular client. See **context handle.**

**runtime semantics.** RPC: The rules of run time for a remote procedure call, including the effect of multiple calls on the outcome of a procedure’s operation. See **at-most-once semantics, broadcast semantics, idempotent semantics, and maybe semantics.**

**S**

**scalability.** The ability of a distributed system to expand in size without changes to the system structure, applications, or the way users deal with the system.

**schema.** See **directory schema.**

**secret key.** Security: A long-lived encryption key shared between a principal and the Authentication Service.

**Security Service.** A DCE component that provides trustworthy identification of users, secure communications, and controlled access to resources in a distributed system.

**segment.** One or more contiguous elements of a string.

**server.** (1) On a network, the computer that contains programs, data, or provides the facilities that other computers on the network can access. (2) The party that receives remote procedure calls. Contrast with **client.**

**server addressing information.** RPC: An RPC protocol sequence, network address, and endpoint that represent one way to access an RPC server over a network; a part of server binding information. See **network address.** See also **binding information, endpoint, and RPC protocol sequence.**

**server application thread.** RPC: A thread running the server application code that initializes the server and listens for incoming calls. See **application thread, client application thread, local application thread, and RPC thread.**

**server binding information.** RPC: Binding information for a particular RPC server. See **binding information and client binding information.**

**server entry.** RPC: A name service entry that stores the binding information associated with the RPC interfaces of a particular RPC server and object Universal Unique Identifiers (UUIDs) for any objects offered by the server. See also **binding information, NSI binding attribute, NSI object attribute, object and RPC interface.**

**server instance.** RPC: A server running in a specific address space. See **server.**

**server state.** Application Support Server: The condition of the Application Support Server after it has been started. The server state may be any of the following, depending on the actions directed to it by the administrator: initializing, quiescent, starting, operating, or stopping.

**server stub.** RPC: 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 **client stub.** See also **manager.**

**service.** In network architecture, the capabilities that the layers closer to the physical media provide to the layers closer to the end user.

**service ticket.** Security: A ticket for a specified service other than the ticket-granting service. See **privilege ticket, simple ticket, and ticket-granting ticket.**

**session.** GDS: A sequence of directory operations requested by a particular user of a particular directory user agent (DUA) using the same session object management (OM) object.

**session key.** Security: A short-lived encryption key provided by the Authentication Service to two principals for the purpose of ensuring secure communications between them. Synonymous with **conversation key.**
shell script. A file containing shell commands. If the file can be processed, you can specify its name as a simple command. Processing of a shell script causes a shell to run the commands in the script. Alternatively, a shell can be requested to run the commands in a shell script by specifying the name of the shell script as the operand `sh` utility.

**SID.** Subject identifier.

**signal.** Threads: To wake only one thread waiting on a condition variable. See broadcast.

**signed.** Security: Pertaining to information that is appended to an enciphered summary of the information. This information is used to ensure the integrity of the data, the authenticity of the originator, and the unambiguous relationship between the originator and the data.

**sign-on.** (1) A procedure to be followed at a terminal or workstation to establish a link to a computer. (2) To begin a session at a workstation. (3) Same as log on or log in.

**simple name.** CDS: One element in a CDS full name. Simple names are separated by slashes in the full name.

**simple ticket.** Security: A ticket that contains the principal’s identity, a session key, a timestamp and other information, sealed using the target’s secret key. See privilege ticket, service ticket, and ticket-granting ticket.

**single sign-on.** In z/OS DCE, single sign-on to DCE allows a z/OS user who has already been authenticated to an MVS external security manager, such as RACF, to be logged in to DCE. DCE does this automatically when a DCE application is started, if the user is not already logged in to DCE.

**skew.** The time difference between two clocks or clock values.

**socket.** A unique host identifier created by the concatenation of a port identifier with a TCP/IP address.

**specific.** XOM: The attribute types that can appear in an instance of a given class, but not in an instance of its superclasses.

**SPI.** (1) System programming interface. (2) Stub programming interface.

**S-stub.** GDS: The part of the directory system agent (DSA) that establishes the connection to the communications network.

**standard.** A model that is established and widely used.

**string.** An ordered sequence of bits, octets, or characters, accompanied by the string’s length.

**stub.** RPC: A code module specific to an RPC interface that is generated by the Interface Definition Language (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. See client stub and server stub.

**Stub programming interface (SPI).** RPC: A private runtime interface whose routines are unavailable to application code.

**subject identifier (SID).** A string that identifies a user or set of users. Each SID consists of three fields in the form person.group.organization. In an account, each field must have a specific value; in an access control list (ACL) entry, one or more fields may use a wildcard.

**synchronization.** DTS: The process by which a Distributed Time Service entity requests clock values from other systems, computes a new time from the values, and adjusts its system clock to the new time.

**syntax.** (1) XOM: An object management (OM) syntax is any of the various categories into which the OM specification statically groups values on the basis of their form. These categories are additional to the OM type of the value. (2) A category into which an attribute value is placed on the basis of its form. See attribute syntax.

**System programming interface (SPI).** A private interface reserved for use by other services within a system and not available to application code. Contrast with API.

**system time.** The time value maintained and used by the operating system.

**T**

**TCP.** Transmission Control Protocol

**TCP/IP.** Transmission Control Protocol/Internet Protocol

**TDF.** Time differential factor.

**thread.** A single sequential flow of control within a process.

**thread handle.** RPC: A data item that enables threads to share a storage management environment.

**Threads Service.** A DCE component that provides portable facilities that support concurrent programming.
The threads service includes operations to create and control multiple threads of execution in a single process and to synchronize access to global data within an application.

ticket. Security: An application-transparent mechanism that transmits the identity of an initiating principal to its target. See privilege ticket, service ticket, simple ticket and ticket-granting ticket.

ticket-granting ticket. Security: A ticket to the ticket-granting service. See privilege ticket, service ticket, and simple ticket.

time differential factor (TDF). DTS: The difference between universal time coordinated (UTC) and the time in a particular time zone.

timeout period. The amount of time in seconds that the Control Task waits for a daemon to initialize successfully. If this time elapses and the daemon does not indicate to the Control Task that it has successfully initialized, the daemon’s state is deemed to be UNKNOWN.

time provider (TP). DTS: A process that queries universal time coordinated (UTC) from a hardware device and provides it to the server.

time provider interface (TPI). An interface between the DTS server and external time provider process. The DTS server uses the interface to communicate with the time provider and to obtain timestamps from an external time source.

time provider program. DTS: An application that functions as a time provider.

time slicing. A mechanism by which running threads are preempted at fixed intervals. This ensures that every thread is allowed time to be executed.

top level pointer. RPC: A pointer parameter that in a chain of pointers is the only member that is not the referent of any other pointer.

tower. CDS: A set of physical address and protocol information for a particular server. CDS uses this information to locate the system on which a server resides and to determine which protocols are available at the server. Tower values are contained in the CDS_Towers attribute associated with the object entry that represents the server in the cell namespace.

TP. Time provider.

TP server. DTS: A server connected to a time provider (TP).

TPI. Time provider interface.

transaction. (1) A unit of processing consisting of one more application programs initiated by a single request, often from a terminal. (2) IMS/ESA: A message destined for an application program.

transfer syntax. RPC: A set of encoding rules used for transmitting data over a network and for converting application data to and from different local data representations. See also Network Data Representation.

Transmission Control Protocol (TCP). A communications protocol used in Internet and any other network following the U.S. Department of Defense standards for inter-network protocol. TCP provides a reliable host-to-host protocol in packet-switched communication networks and in an interconnected system of such networks. It assumes that the Internet Protocol is the underlying protocol. The protocol that provides a reliable, full-duplex, connection-oriented service for applications.


transmitted type. RPC: For data types with the 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. See presented type.

transport independence. RPC: 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. See transport layer and transport protocol.

transport layer. A network service that provides end-to-end communications between two parties, while hiding the details of the communications network. The Transmission Control Protocol (TCP) and International Organization for Standardization (ISO) TP4 transport protocols provide full-duplex virtual circuits on which delivery is reliable, error free, sequenced, and duplicate free. User Datagram Protocol (UDP) provides no guarantees. The connectionless RPC protocol provides some guarantees on top of UDP.

transport protocol. A communications protocol, such as the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).

trust peer. One side of a cross-registration that enables two cells to have peer trust. See peer trust.

type. XOM: A category into which attribute values are placed on the basis of their purpose. See attribute type.
**type UUID**. RPC: The Universal Unique Identifier (UUID) that identifies a particular type of object and an associated manager. See also *manager* and *object*.

**V**

**VFS.** Virtual file system.

**Virtual file system (VFS).** DFS: A level of abstraction above the specific interfaces to various types of file systems. It is used to avoid having to change kernel code to handle low-level, system-specific differences.

**Virtual Telecommunications Access Method (VTAM®).** An IBM licensed program that controls communication and the flow of data in an SNA network. It provides single-domain, multiple-domain, and interconnected network capability.

**VTAM.** Virtual Telecommunications Access Method.

**W**

**well-known endpoint.** RPC: A preassigned, stable endpoint that a server can use 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. See *dynamic endpoint* and *endpoint*.

**workstation.** A device that enables users to transmit information to or receive information from a computer, for example, a display station or printer.

**X**

**X.500.** The CCITT/ISO standard for the open systems interconnection (OSI) application-layer directory. It allows users to register, store, search, and retrieve information about any objects or resources in a network or distributed system.

**XDS.** The X/Open Directory Services API.

**X/Open Directory Services (XDS).** An application program interface that DCE uses to access its directory service components. XDS provides facilities for adding, deleting, and looking up names and their attributes. The XDS library detects the format of the name to be looked up and directs the calls it receives to either GDS or CDS. XDS uses the XOM API to define and manage its information.

**XOM.** The X/Open OSI-Abstract-Data Manipulation API.
Bibliography

This bibliography is a list of publications for z/OS DCE and other products. The complete title, order number, and a brief description is given for each publication.

z/OS DCE Publications

This section lists and provides a brief description of each publication in the z/OS DCE library.

Overview

- **z/OS DCE Introduction**, GC24-5911
  This book introduces z/OS DCE. Whether you are a system manager, technical planner, z/OS system programmer, or application programmer, it will help you understand DCE and evaluate the uses and benefits of including z/OS DCE as part of your information processing environment.

Planning

- **z/OS DCE Planning**, GC24-5913
  This book helps you plan for the organization and installation of z/OS DCE. It discusses the benefits of distributed computing in general and describes how to develop plans for a distributed system in a z/OS environment.

Administration

- **z/OS DCE Configuring and Getting Started**, SC24-5910
  This book helps system and network administrators configure z/OS DCE.

- **z/OS DCE Administration Guide**, SC24-5904
  This book helps system and network administrators understand z/OS DCE and tells how to administer it from the batch, TSO, and shell environments.

- **z/OS DCE Command Reference**, SC24-5909
  This book provides reference information for the commands that system and network administrators use to work with z/OS DCE.

- **z/OS DCE User's Guide**, SC24-5914
  This book describes how to use z/OS DCE to work with your user account, use the directory service, work with namespaces, and change access to objects that you own.

Application Development

- **z/OS DCE Application Development Guide: Introduction and Style**, SC24-5907
  This book assists you in designing, writing, compiling, linking, and running distributed applications in z/OS DCE.

- **z/OS DCE Application Development Guide: Core Components**, SC24-5905
  This book assists programmers in developing applications using application facilities, threads, remote procedure calls, distributed time service, and security service.

- **z/OS DCE Application Development Guide: Directory Services**, SC24-5906
  This book describes the z/OS DCE directory service and assists programmers in developing applications for the cell directory service and the global directory service.

- **z/OS DCE Application Development Reference**, SC24-5908
  This book explains the DCE Application Program Interfaces (APIs) that you can use to write distributed applications on z/OS DCE.

Reference

- **z/OS DCE Messages and Codes**, SC24-5912
  This book provides detailed explanations and recovery actions for the messages, status codes, and exception codes issued by z/OS DCE.

z/OS SecureWay® Security Server Publications

This section lists and provides a brief description of books in the z/OS SecureWay Security Server library that may be needed for z/OS SecureWay Security Server DCE and for RACF® interoperability.
• **z/OS SecureWay Security Server DCE Overview**
  GC24-5921
  This book describes the z/OS SecureWay Security Server DCE and provides z/OS SecureWay Security Server DCE information about the z/OS DCE library.

• **z/OS SecureWay Security Server LDAP Client Programming**
  SC24-5924
  This book describes the Lightweight Directory Access Protocol (LDAP) client APIs that you can use to write distributed applications on z/OS DCE and gives you information on how to develop LDAP applications.

• **z/OS SecureWay Security Server RACF Security Administrator's Guide**
  SA22-7683.
  This book explains RACF concepts and describes how to plan for and implement RACF.

• **z/OS SecureWay Security Server LDAP Server Administration and Use**
  SC24-5923
  This book describes how to install, configure, and run the LDAP server. It is intended for administrators who will maintain the server and database.

• **z/OS SecureWay Security Server Firewall Technologies**
  SC24-5922
  This book provides the configuration, commands, messages, examples and problem determination for the z/OS Firewall Technologies. It is intended for network or system security administrators who install, administer and use the z/OS Firewall Technologies.

---

**Tool Control Language Publication**

• *Tcl and the Tk Toolkit*, John K. Osterhout, (c)1994, Addison—Wesley Publishing Company.
  This non-IBM book on the Tool Control Language is useful for application developers, DCECP script writers, and end users.

---

**IBM C/C++ Language Publication**

• **z/OS C/C++ Programming Guide**
  SC09-4765
  This book describes how to develop applications in the C/C++ language in z/OS.

---

**z/OS DCE Application Support Publications**

This section lists and provides a brief description of each publication in the z/OS DCE Application Support library.

• **z/OS DCE Application Support Configuration and Administration Guide**
  SC24-5903
  This book helps system and network administrators understand and administer Application Support.

• **z/OS DCE Application Support Programming Guide**
  SC24-5902
  This book provides information on using Application Support to develop applications that can access CICS® and IMS™ transactions.
Encina Publications

- **z/OS Encina Toolkit Executive Guide and Reference** SC24-5919
  This book discusses writing Encina applications for z/OS.

- **z/OS Encina Transactional RPC Support for IMS** SC24-5920
  This book is to help software designers and programmers extend their IMS transaction applications to participate in a distributed, transactional client/server application.
Index

Special Characters
.profile file
   using to set variables 55

A
access checking, model 180
access control 163
Access Control List (ACL)
   application role in interface 183
   authorization 180
   backing store routines 182
   editor 219
   entries
      and access checking model 180
      any_other 181
      foreign_group 181
      foreign_other 181
      foreign_user 181
      group 181
      other_object 181
      types 180
      user 181
   example 131
   extracting and evaluating 186
   manager 182
      definition 185
      example 188
      prinstrings 185
   permissions 180
   privilege attributes 180
   remote interface (rdacl) 182
   requirements on application to use 183
   routines 182
ACF (Attribute Configuration File)
   See Attribute Configuration File (ACF)
ACL (Access Control List)
   See Access Control List (ACL)
adding multiple object type, Greet example 100
administration, DCE
   DCE control program (dcecp) 10
ALL31 57
allocating data sets 32
allocation, memory (RPC) 249
application
   building
      in batch 50
      in the shell 48
   code
      link-editing 45
   example 273
   interface, naming 37
application (continued)
   Language Environment
      interlanguage calls 58
      runtime options 57
   link-editing 45
   multithreaded 150
   producing an executable 274
   server 39
   status
      checking in batch 52
      checking in the shell 52
      checking in TSO/E 52
   stopping
      abnormally 52
      from the shell 52
      from TSO/E 52
      normally 52
application development, DCE
   application role in security 183
   DCE control program (dcecp) 10
   DCE Host Daemon 9
   declaring data types and constants for 12
   example 273
   interface definition
      handle 8, 9
   interface routines, overview 10
   multithreaded applications 150
   object, definition of 184
   overview 4
   parallelism 150
   steps 5
   steps, high level 29
   threads, using 149
   tools
      attribute configuration language 7
      interface definition language 7
      interdomain interface definition language (IDL) 8
      remote DCE management 7
      UUID generator (uuidgen) 7
application example, DCE 273
application, sample
   client code 272
   makefile 271
   manager code 272
   object bind interface 272
   server code 271
arrays, RPC 255
at-most-once (RPC execution semantic) 241
Attribute Configuration File (ACF)
   description 37
   file
      timop 275, 285
Attribute Configuration File (ACF) (continued)
  Greet example, auto_handle 98
  overview 9
attributes, privilege 180
audience skills xvii
authentication
  and binding methods 213
authentication, security
  authenticated RPC 169
  client, steps 174
  identity 168
  identity, obtaining 168
  keys 169
  keytab files 169
  model 165
  server, steps 170
  ticket-granting ticket (TGT) 168
authorization, security
  ACLs 180
  manager, example 188
  managers 182
  overview 176
  permissions 180
auto_handle, ACF, Greet example 98
automatic binding 209
  client code, Greet application 99
  handle, using 97
  management 97
  manager code, Greet example 99
  modifying the Greet application 98
auxiliary header file, timop 285

B
backing store
  ACL UUID-indexed
    contents 187
  name-indexed
    contents 187
  object-indexed
    data type 187
    header contents 187
batch
  building applications in 50
  checking application status in 52
  compiling client and server programs in 43
  compiling IDL files in 39
dcelogin 75
link-editing in 47
running client application in 51
running server application in 50
runtime environment variables
  setting in 56
UUID
  generating in 36

BELOWHEAP 57
bibliography 353
binding
  authentication and 213
call routing 206
definition of 203
  handles
    customized 212
    data type (in RPC) 212
    primitive 212
importing and exporting 221
information
  Greet application, exporting 86
  information repository (NSI) 215
  information, components of 203
  information, models for accessing
    by objects 232
    by servers 232
    by services 231
interface, sample code for 272
management
  automatic method 97
  methods
    explicit 210
    implicit 210
model, client 205
model, server 204
name service role in 217
object UUIDs as endpoint identifiers 208
overview 203
role of DNS 221
routing policy 208
RPC protocol version identifier (binding information) 203
semantics 212
transfer syntax identifier (binding information) 203
body, interface 12
books, list of DCE and related 353
broadcast (RPC execution semantic) 242
building TIMOP 278

C
C language
  data types
    mapping to IDL types 245
C programs
  compiling with DLL option 40
c89 compiler
  compiling 41
  link-editing 45
call routing 206
call, remote procedure 9
cataloged procedure
  EDCC 43
  EDCPL 47
CC EXEC  42
CDS (Cell Directory Service)  
See Cell Directory Service (CDS)
Cell Directory Service (CDS)
accessing  219
entry attributes  219, 220
entry types
  binding  219
  group  219
  object  219
  profile  220
exporting the Greet server binding  87
Greet client source code  92
Greet server source code  88
importing the Greet server binding  91
starting the Greet client  95
starting the Greet server  94
use in NSI to access  219
XDS interface to  219
character parameter, RPC  248
checking application status  52
client
  application
    running in batch  51
    running in the shell  51
    running in TSO/E  51
  authenticating  174
  binding model  205
code
  Greet application  66
  Greet application, automatic binding  99
  writing  40
control
  managing server with  267
  credentials, security  179
generic, sample code for  272
  Greet
    output  95
  programs, compiling
    in batch  43
    in the shell  41
    in TSO/E  42
  role in security  163
  source
    file, timop  275
    listing, timop  287
  starting  51
  use of name service, example  221
CLIST
  CPLINK  46
commands
  dcelogin  75
  server show  10
  server start  10
tso dcelogin  75
  tso netstat int  96
commands (continued)
  uidgen  11
compile options  45
compiling
  client and server programs
    in batch  43
    in the shell  41
    in TSO/E  42
  DCE applications  40
  DCE applications, sample JCL  44
  Greet interface  61
  IDL file
    in batch  39
    in the shell  38
    in TSO/E  39
  interface  37
context handles (RPC)  255
Control Program, DCE (dcecp)  
functions  10
CPARM option  45
CPLINK CLIST  46
creating
  Greet application  58
  IDL file  35, 36
credentials, security  179
customized binding handle  212
customizing
  environment, runtime  53
  runtime environment  53
D
daemon, DCE Host (dced)  9
data protection  163
data sets
  allocating  32
  Greet application  59
data types
  arrays  255
  character  248
  context handles  255
  handle_t  212
  IDL to C type mappings  245
  marshalling  262
  pipes  261
  pointers  248
  rpc_binding_handle_t  212
  structures and unions  259
database, backing store  186
DBCS data  248
DCE (Distributed Computing Environment)  48
DCE application development
  See application development, DCE
DCE application files  30, 31
  system-generated  31
  user-written  30
DCE Control Program
functions 10

DCE file names
example of remote 215
junctions in 215

DCE Host Daemon
functions 9

DCE Name Service Interface (NSI)
See Name Service Interface (NSI), DCE

DCE security
See security, DCE
dce_acl_is_client_authorized 185
dce_db_std_header_init 187
dce_server_inq_server 267
dce_server_register 268
dce_server_sec_begin 268
DCE, logging in 75
dcecp
See Control Program, DCE (dcecp)
dcecp and server information 9
dced (DCE Host Daemon)
functions 9
dced_server_disable_if 267
dced_server_enable_if 267
dcelogin
command 75
in batch 75
debugging, and dcecp 10
defining
interface operations 37
definition side-deck 40
developing applications 29
developing DCE applications 273
development, application
See application development, DCE

DFS (Distributed File Service)
See Distributed File Service (DFS)

Directory Services, X/Open (XDS) interface
See X/Open Directory Services (XDS) interface

Distributed Computing Environment (DCE)
application
building in batch 50
building in the shell 48
running 50
simplified development cycle 273
starting 50
client, running sample JCL 51
DTS operations, example application 276
server application
running in batch 50
running in the shell 50
server, sample JCL to run 51

Distributed File Service (DFS)
junctions 215

DLL (Dynamic Link Library)
See also Dynamic Link Library (DLL)

DLL (Dynamic Link Library) (continued)
support of C/C++ compiler 40
dynamic endpoint 223
Dynamic Link Library (DLL)
compiler option 40
link-editing 40

E
EDCC cataloged procedure 43
EDCPL cataloged procedure 47
endpoint
binding information 203
identifiers 208
map 9
obtaining 223
registering the Greet server 82
Entry Point Vector (EPV) 14
envar file
using to set variables 56
environment, runtime
variables 53
setting in batch 56
setting in the shell 55
setting in TSO/E 56
EPM, using, Greet application 79
EPV (Entry Point Vector)
See Entry Point Vector (EPV)
examples
application code 271
application, DCE DTS operations 276
application, Greet 58
application, timop 273
IDL file 34
monitoring 96
EXEC
CC 42
executable DCE application, producing 274
explicit binding 210
export command
using to set variables 55
exporting binding information
Greet example 86
exporting bindings 221
exporting Greet server binding 87

F
file
for DCE application 30, 31
file names, DCE
example of remote 215
junctions in 215

File Service, Distributed (DFS)
See Distributed File Service (DFS)
forking in a threaded program 161
forwarding 223
free_inst (sample) 264
from_xmit (sample) 265
full pointer parameters (RPC) 243

G
generating IDL file 35
generic RPC functions 39
glossary 333
Greet client
code for CDS 92
code, multiple objects 111
multiple objects 110
output 74, 95
starting 73
starting, using CDS 95
Greet example
ACF, auto_handle 98
application 58
automatic binding
client code 99
IDL file 98
manager code 99
building 69
client code 66
compiling the interface 61
creating files 59
data sets 59
defining interface operations 61
exporting binding information 86
extending 75
fast path 58
generating IDL file 59
generating UUID 59
header file 63
link-editing 71
manager code 63
modifying the Greet Server Code 79
multiple object type, adding 100
name-based authorization 117
naming the interface 60
object type management 101
PAC-based 131
searching for server 78
securing 117
server code 63
server entry 86
using automatic binding 98
using the EPM 79
using the Name Service Interface (NSI) 86
Greet IDL file, object type management 102
Greet manager, multiple object type 103
Greet server
binding 87, 91
exporting 87

Greet server (continued)
binding (continued)
importing 91
code, for CDS 88
multiple objects 104
output 95
starting 72
starting, using CDS 94
Greet server endpoint, registering 82
group entry, directory 229

H
handle
interface 8
handle, binding
See also binding, handles
methods
automatic 209
parameters
binding handles as 209
header
client, timop 286
header file
Greet application 63
header, interface 12
file
parts 12
HEAP 57
host
daemon (dced) 9
data
controlling 9
Host Daemon, DCE
functions 9

I
identifier, object (OID)
use in directory service 220
identity
checking 164
principal (security) 165
IDL (Interface Definition Language)
See Interface Definition Language (IDL)
IDLALLOC REXX EXEC 33
implicit binding 210
importing
bindings 221
Greet server binding 91
initialization, server 13
initializing server manager 268
instance, server 9, 203
interface
body 12
definition
generating with uuidgen 11

Index 361
interface (continued)
definition (continued)
  RPC 8
tentry point vector (EPV) 14
handle 8
header 12
naming the application 37
operation declaration 12
registration 14
routines, application development 10
specification 8
Interface Definition Language (IDL)
cataloged procedure 39
compiler
  output 37
  running 39
  sample JCL 39
compiler output 37
compiler, shell 38
file
  compiling in batch 39
  compiling in shell 38
  compiling in TSO/E 39
example 34
generating 273
generating in batch 36
generating in the shell 35
generating in TSO/E 35
generating with uuidgen 11
  Greet example, automatic binding 98
timop 284
operations 12
REXX EXEC 39
interface operations, defining 37
interface uuid 216
interface, ambiguous 224
interlanguage calls 58
international character data type (RPC) 248

J
junction, namespace 218
  junction, namespace
  and ACL editor 219
K
kerberos 165
key 170
keytab files 169
L
Language Environment (continued)
  application storage storage options
    ALL31 57
    BELOWHEAP 57
    HEAP 57
    LIBSTACK 57
    STACK 57
    STORAGE 57
  runtime options 57
LIBSTACK 57
link-editing
  application code
    in batch 47
    in the shell 45
    in TSO/E 46
  clients, sample JCL 47
  DCE applications 40, 45
  DCE servers, sample JCL 48
  Greet example 71
  pre-link step 45
logging into DCE 75
login
  context and security 168
LONGNAME option 45
lookups, name 229
lower-level pointer, definition 250
M
makefile sample code 271
managing routine (sample) 171
management
  services, DCE
    remote, providing 9
manager
  ACL 182
  ACL, example 188
    generic, sample code for 272
manager code
  Greet application 63
  Greet example
    automatic binding 99
    writing 39
manager routine 39
manager source listing, timop 299
map, endpoint 9
mapping 223
marshalling
  data 243
    data types (RPC) 262
maybe (RPC execution semantic) 242
memory allocation (RPC) 243, 249
monitoring distributed applications 96
multiple objects
  Greet client 110
  Greet client code 111
multiple objects (continued)
  Greet server 104
  starting the Greet4 application 114
  type, Greet example 100
multithreading 14, 150

N
name lookups, in directory 229
name resolution, server object 183
Name Service Interface (NSI), DCE
  binding 217
    ambiguous 224
    binding handle, contents 222
    endpoint mapper 223
    importing and exporting bindings 221
    partial 224
    partial binding 223
    binding, partial and 224
  Cell Directory Service (CDS) 219
  client and server use of, example 221
  Greet application 86
  group entry 229
  junction 215
  junction, namespace
    example 218
  names versus objects 215
  namespace 215
  namespace, object-oriented 226
  objects, examples of 226
  OIDs 220
  overview 215
  profile entry 230
  terminology 219
  use as binding repository 215
  use of CDS as database 219
  uuid 216
name-based authorization
  example 117
names, file, DCE
  example of remote 215
  junctions in 215
namespace
  entry types
    profile 231
    server 231
  exporting Greet server binding 87
  importing Greet server binding 91
  object-oriented 226
naming the application interface 37
network monitor
  dynamic version 96
non-pointer parameters (RPC) 243
NSI (Name Service Interface)
  See Name Service Interface (NSI), DCE
O
object
  databases with information about 184
  definition 184
  state information 184
object identifier (OID)
  use in directory service 220
object type management
  Greet application 101
  Greet IDL file 102
  Greet manager 103
object UUID 208, 216
object-oriented namespace
  setting up 227
object, definition of a DCE 216
objects, directory, examples of 226
on-demand server 267
operation declaration 12
output
  Greet client 95
  Greet server 95
P
PAC-based security
  example 131
parallelism 150
password, security 170
permissions, security 180
persistent server 267
pinging a server 10
pipes (RPC) 261
  data types
    marshalling 262
pointers (RPC) 248
portability, application 3
principal identity, security 165
privilege attributes 180
privilege ticket-granting ticket (PTGT) (security) 168
PROC
  EDCC 43
  EDCPL 47
  IDL 39
  UUIDGEN 36
profile entry, directory 230
  binding
    information, models for accessing 231
    information, models for accessing (non-CDS) 233
    namespace
      global organization of 238
    parameters
      overview 241

Index 363
Remote Procedure Call (RPC) (continued)

data types (continued)
  constructed 244
  context handles 255
  IDL to C type mappings 245
  pipes 261
  pointers 248
  structures and unions 259

execution semantics
  attributes 241
failure
  states 242
finding servers 203
functions, generic 39
interface body 12
interface handle 8
interface header 12
interface operation declaration 12
memory
  allocation, client side 243
  management 243
  memory allocation 249
parameter semantics 242
parameters
  character 248
  context handles 255
  in,out type 244
  memory management 243
  out type 244
  pointers 248
  semantics 242
  status 243
  structures and unions 259
pointers
  lower-level 250
  rules for using 250
  top-level 250
types 249
protocol version identifier (binding information) 203
server instance 9
servers (endpoints)
  mapper 9

RENT option 45
repository, binding, use of namespace as 215
resolver routine 186

REXX EXEC
  IDL 39
  IDLALLOC 33
  UUIDGEN 35

routines, application development interface 10
routing policy 208

RPC (Remote Procedure Call)
  See Remote Procedure Call (RPC)

rpc_binding_inq_auth_caller 164
authentication
  model 165
security, DCE (continued)
authorization (continued)
   ACL manager, example 188
   ACL managers 182
   credentials 179
   overview 176
   permissions 180
example 117
kerberos 165
model 163
overview 163
principal 165
protocol, shared secret key 165
semantics, binding 212
semantics, execution, RPC 241
server
   authenticating 170
   binding model 204
   code
      Greet application 63
      requirements to use ACL management 183
      writing 39
   commands
      server show 10
      server start 10
   connecting (binding) to a 203
   controlling
      remotely 9
   generic, sample code for 271
   Greet, output 95
   header file listing, timop 295
   host network address (binding information) 203
   identity, checking 164
   initialization 13
   instance 9
      endpoint of (binding information) 203
   management
      application support for 268
      overview 267
   manager
      initialization 268
   managing with a control client
      example 267
   multithreading 14
   object name resolution 183
   pinging 10
   programs, compiling
      in batch 43
      in the shell 41
      in TSO/E 42
   registration 14
   role in security 163
   source file, timop 275
   source listing, timop 296
   starting 50
      by dced 267
      in batch 50

sample JCL
   compiling applications 44
   IDL compiler 39
   link-editing DCE servers 48
   running a DCE client 51
   running a DCE server 51
   to link-edit DCE clients 47
sample_client.c sample code 272
sample_manager.c sample code 272
sample_server.c sample code 271
scripts 10
SDSF panel 52
searching for server, Greet application 78
sec_acl routines 182
sec_key_mgmt_get_key 168
sec_key_mgmt_manage_key 170
sec_login_setup_identity 168
sec_login_validate_identity 168
secret key protocol 165
security, DCE
   application program roles in 165
   application role in 183
   authentication
      authenticated RPC 169
      client, steps 174
      identity, obtaining 168
      keys, managing 169
      keytab files 169
      server, steps 170
      ticket-granting ticket (TGT) 168
   authorization
      Access Control Lists (ACLs) 180

Index 365
server (continued)
  starting (continued)
    in the shell 50
  types
    on-demand 267
    persistent 267
  use of name service, example 221
server_get_identity 171
server_renew_identity 172, 173
shell
  building applications in 48
  checking application status in 52
  commands
    export 55
  compiling client and server programs in 41
  compiling IDL files in 38
  link-editing in 45
  running client application in 51
  runtime environment variables
    setting in 55
  starting server application in 50
  stopping application from 52
  using envar file 56
UUID
  generating in 35
side-deck, definition 40
simplified development cycle, DCE application 273
skeleton IDL file 273
skills, for audience xvii
source file timop 273
source listing, timop reference monitor 301
STACK 57
starting
  client 51
  DCE application 50
  Greet client 73
  Greet client, for CDS 95
  Greet example, multiple objects 114
  Greet server 72
  Greet server, for CDS 94
  server 50
state information, object 184
stopping
  applications 52
  Greet example 74
STORAGE 57
storage allocation (RPC) 243
store, backing
  ACL UUID-indexed
    contents 187
  name-indexed
    contents 187
  object-indexed
    data type 187
    header contents 187
structures (RPC) 259
stub file
  purpose 276

T
TCP/IP Real Time Network Monitor 96
terminating threads 157
terminology
  in this book xx
TERMTHDACT 57
threads
  multithreading 150
  overview 149
  policy, scheduling 151
  policy, use 150
  programming topics
    cancelation points 156
    canceling threads 155
    forking in a threaded program 161
    signals 159
    storage for thread specific data 154
    terminating threads 157
    thread cleanup 158
  related routines
    pthread_server_listen 151
  routines
    pthread_cancel 155
    pthread_cleanup_pop 158
    pthread_cleanup_push 158
    pthread_cond_signal 161
    pthread_cond_timedwait 156
    pthread_cond_wait 153
    pthread_create 149
    pthread_delay_np 156
    pthread_detach 154
    pthread_equal 154
    pthread_exit 158
    pthread_getspecific 154
    pthread_join 154
    pthread_keycreate 154
    pthread_mutex_lock 152
    pthread_mutex_unlock 152
    pthread_setspecific 154
    pthread_testcancel 156
  RPC
    semantics 162
    rules for using 152
    safety
      blocking behavior 152
      problem contexts 152
Index

threads (continued)
safety (continued)
reentrancy 152
ticket-granting ticket (TGT) (security) 168
timpan
ACF file 275
ACF file listing 285
auxiliary header file listing 285
client
description 276
header, source listing 286
source listing 287
client source file 275
description 276
example application 273
IDL file listing 284
manager
source listing 299
reference monitor
source listing 301
running 279, 280
server
description 276
header file listing 295
source listing 296
server source file 275
source code listing 284
source file 273
source files 277
suggested enhancements 282
tools
endpoint mapper 9
top-level pointer, definition 250
TRACE 57
transfer syntax identifier (binding information) 203
transmit_as attribute (RPC) 262
tso dcelogin command 75
tso netstat int command 96
TSO/E
checking application status in 52
compiling client and server programs in 42
compiling IDL files in 39
link-editing in 46
running client application in 51
runtime environment variables
setting in 56
stopping application from 52
UUID

generating in 35
type manager
Greet application 101
Greet client 110
Greet IDL file 102
Greet manager 103
Greet server 104
typedefs 247

U
unions (RPC) 259
unique identification 7
Universal Unique Identifier (UUID)
generating in batch 36
generating in the shell 35
generating in TSO/E 35
generator 7
identifying interfaces 37
interface 216
object 216
object, and binding ambiguity 225
object, use of, example 225
use of OID instead of 220
uses in DCE 216
UNIX System Services
parameter considerations
MAXTHREADS 58
MAXTHREADLASTASKS 58
utilities, DCE
uuidgen 11
UUID (Universal Unique Identifier)
See Universal Unique Identifier (UUID)
UUIDGEN cataloged procedure 36
uuidgen command 273
UUIDGEN REXX EXEC 35
uuidgen utility 35

V
variables
environment, runtime
_EUV_EXC_ABEND_DUMPS 54
_EUV_EXC_SW_DUMPS 54
_EUV_RPC_COMM_TIMEOUT 54
_EUV_RPC_DYNAMIC_POOL 53
_EUV_SVC_API_DUMPS 53
LANG 53
NLSPATH 53
overview 53
RPC_DEFAULT_ENTRY 53
RPC_DEFAULT_ENTRY_SYNTAX 53
setting in batch 56
setting in the shell 55
setting in TSO/E 56
setting with .profile file 55

X
X/Open Directory Services (XDS) interface 219
XDS (X/Open Directory Services) interface
See X/Open Directory Services (XDS) interface
You may use this form to report errors, to suggest improvements, or to express your opinion on the appearance, organization, or completeness of this book.

Date: _____________________________________________________________________

__________________________________________________________________________

__________________________________________________________________________

__________________________________________________________________________

__________________________________________________________________________

__________________________________________________________________________

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Note

Report system problems to your IBM representative or the IBM branch office serving you. U.S. customers can order publications by calling the IBM Software Manufacturing Solutions at 1-800-879-2755.

In addition to using this postage-paid form, you may send your comments by:

FAX 1-607-752-2327 Internet pubrcf@vnet.ibm.com
IBMLink GDLVME(PUBRCF)

Would you like a reply? YES NO If yes, please tell us the type of response you prefer.

Electronic address: ____________________________
FAX number: ________________________________
Mail: (Please fill in your name and address below.)

Name __________________________________________ Address ______________________________
Company or Organization ______________________________ Phone No. _________________________