OSF® DCE Application Development Guide —Core Components Release 1.2.2

December 10, 1998

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142

IBM

OSF® DCE Application Development Guide —Core Components

Release 1.2.2

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

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

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

Copyright © 1995, 1996 Open Software Foundation, Inc.

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

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

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

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

Copyright © 1990, 1991 Siemens Nixdorf Informationssysteme AG

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

Copyright © 1988, 1989, 1995 Massachusetts Institute of Technology

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

Copyright © 1995, 1996 Hitachi, Ltd.

All Rights Reserved

Printed in U.S.A.

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

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

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

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

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

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

DECstation 3100 and DECnet are trademarks of Digital Equipment Corporation.

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

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

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

DFS is a trademark of the Transarc Corporation.

Episode is a registered trademark of the Transarc Corporation.

Ethernet is a registered trademark of Xerox Corporation.

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

IBM is a registered trademark of International Business Machines Corporation.

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

MX300i is a trademark of Siemens Nixdorf Informationssysteme AG.

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

PostScript is a trademark of Adobe Systems Incorporated.

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

NetWare is a registered trademark of Novell, Inc.

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

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

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

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

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

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

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

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

Contents

Figures

Tables

Preface

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

Audience

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

Applicability

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

Purpose

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

Document Usage

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

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

Related Documents

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

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

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

Typographic and Keying Conventions

This guide uses the following typographic conventions:

Bold	**Bold** words or characters represent system elements that you must use literally, such as commands, options, and pathnames.
Italic	*Italic* words or characters represent variable values that you must supply. *Italic* type is also used to introduce a new DCE term.
`Constant width`	Examples and information that the system displays appear in `constant width` typeface.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard.
...	Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.

This guide uses the following keying conventions:

<Ctrl-x> or ˆx	The notation **<Ctrl-x>** or **ˆx** followed by the name of a key indicates a control character sequence. For example, **<Ctrl-C>** means that you hold down the control key while pressing **<C>**.
<Return>	The notation **<Return>** refers to the key on your terminal or workstation that is labeled with the word Return or Enter, or with a left arrow.

Problem Reporting

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

Pathnames of Directories and Files in DCE Documentation

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

Part 1. DCE Facilities

Chapter 1. Introduction to DCE Facilities

By now you are aware that DCE consists of a number of major components, each of which addresses some necessary aspect of distributed computing: DCE Threads make programs more efficient by allowing parallel execution of portions of code, remote procedure calls (RPCs) hide network details from applications, the DCE Time Service gives consistent time to widely scattered cells and hosts, the DCE Security Service gives programs assurances that users and other programs are who they say they are and that they are authorized to do what they are supposed to do, and the DCE Directory Service helps clients find servers and other resources. For most applications, a DCE component is not used by itself, but the components all work together to create a very useful and powerful environment.

The more you understand DCE and its components, the more you'll realize that a strict division by component is not always clear. The document set for DCE is organized by component mostly for the convenience of people trying to explain and understand DCE, but applications often contain a blend of aspects of all the components. This is why it often seems like the information you need to do your work is scattered across many chapters or volumes, and why advanced or unusual features seem to be described along-side basic or typical tasks. DCE also has some special facilities that just do not fit neatly into any one discussion of a DCE component, and these are the facilities we describe in this first part of the *OSF DCE Application Development Guide—Core Components*.

You should read the *OSF DCE Application Development Guide—Introduction and Style Guide* prior to using the DCE facilities described here, and you may want to skip to other parts of this guide before learning details about the DCE facilities.

Most DCE facilities are already used by one or more major components of DCE to accomplish some feature they require; others are standalone facilities intended to make developing distributed applications easier. These facilities are described separately here so that you can use them for your own applications too. The DCE facilities include the following:

- Host Services

  Host services give remote access to several kinds of data and functionality with respect to each DCE host and its servers. Each host runs a DCE host daemon (**dced**) as the interface to the host services. In many cases, **dced** automatically maintains the data and performs the functions. Some of the data that you can access (and maintain) remotely includes the host name, the host's cell name, configuration and execution data for all servers on the host, and a database of endpoints (server addresses) through which running servers can be contacted. Some of the functions that you can remotely perform include starting and stopping servers.

- Application Message Service

  This service provides a convenient way to manage readable character strings of information that are usually displayed to application users. The service uses message catalogs to maintain message text and explanations separate from the program so that language, cultural, or other site-specific issues are easily managed for applications. The message text can also be in memory during program execution for more efficient programs.

- Serviceability

  Serviceability is another kind of message text service with functionality beyond just the display of general-purpose text. Serviceability is typically used for message text about a server's activity. Messages can be displayed through standard output or standard error, or they can be routed to log files. The serviceability facility maintains message text in catalogs (or memory) just as the application message service does; but, in addition to the text and its explanation, additional attributes specify subcomponents (program modules), message severity, the action users or programs should take, and the debug level.

- Backing Store Database Service

  You use a backing store to maintain typed data between invocations of applications. For example, you could store application-specific configuration data in a backing store, and then, when the application restarts, it could read the previous configuration from the backing store. Data is stored and retrieved by a Universal Unique Identifier (UUID) or character string key, and each record (or data item) may have a standard header if you wish.

As DCE has developed and improved, useful facilities such as serviceability have been added to make DCE easier and more useful. For example, serviceability makes a distributed application much easier to develop. With it, you can log and distinguish debug messages from complex applications involving multiple clients, servers, and threads. Although the major components are required to make DCE work, this kind of facility is not required.

Some solutions developed to implement a major component's feature can also prove useful to your applications. For example, the security component must have a way to maintain access control lists (ACLs). While the backing store was developed to handle this kind of task, you can use this facility to store your own application-specific data across invocations.

This first part of the *OSF DCE Application Development Guide—Core Components* describes how you might put these useful facilities to work in your applications.

Chapter 2. DCE Host Services

Every DCE host must maintain certain kinds of data about itself and the servers it provides. For example, each host stores configuration data about its DCE environment, and it also stores data about servers registered and running on the host. In addition, each host needs some services to not only manage this data but also to administer the host and DCE servers. For example, a service that can start and stop specific servers has obvious value. The DCE host services consist of the following:

- Endpoint Mapper

  The endpoint mapper service enables a client to find servers on a particular host and the services and objects provided by those services. This service maintains on each host an endpoint map that contains a mapping of port addresses (endpoints) to servers, the services servers provide, and the objects servers manage.
- Hostdata Management

  The hostdata management service stores and controls access to such data as the host's cell name, the host name, and the cell alias names, among other things.
- Server Management

  The server management service can start and stop specified servers on a host, enable or disable specific services provided by a server, and manage configuration and execution data about these servers.
- Security Validation

  The security validation service maintains a login context for the host's identity of itself, maintains the host principal's keys, and ensures applications (especially login programs) that the DCE security daemon (**secd**) is genuine.
- Key Table Management

  A server uses private keys for its security instead of human-entered passwords. The key table management service can be used to manage the keys stored in key tables on a server's host.

Of course, in a distributed environment, these data and services must be easily yet securely accessible from other hosts. The DCE host daemon (**dced**) is a continuously running program on each host that provides access to the host services either locally on that host or remotely from another host.

Types of Applications

Although applications may need some aspect of these host services (control over which services are enabled for a particular server, for example), typical servers do not have to do any special coding for them. This reduces the size and complexity of server code and keeps the details of administration out of applications. It also removes the burden of server administration so you can concentrate on the application's business functionality.

System administrators will appreciate this development model too because it is unlikely that many servers implementing their own administrative mechanisms will all behave in the same manner. Administrators commonly use the DCE control program, **dcecp**, to access the host services (via **dced**) of any host in their distributed environment (provided the user has the appropriate permissions). The DCE control program also uses a script language for more sophisticated administration. See the *OSF DCE Administration Guide—Core Components* for more on using **dcecp** to access the host services.

Although **dcecp** commands offer an administrator a great deal of control over DCE hosts and servers, a set of APIs are also supplied for application developers who need to access the DCE host services from an application rather than from scripts or the operating system's command line.

Typical business applications do not use the APIs of these services, but a management application might. A management application is a client or server that manages other servers or some aspect of the distributed environment. (The **dced** program is itself a management application that is built into DCE.) Some other types of applications that might use these API include

- Applications that control other servers for load balancing or server redundancy.
- An application that uses a graphical user interface (GUI) instead of the command-line interface provided by **dcecp**.
- An application that needs to monitor a server's current state. For example, an application may need to make sure a particular server or one of its services is available.

Issues of Distributed Applications

The most important aspect of **dced** is that it gives system administrators the ability to remotely manage services, servers, endpoints, and even objects on any host in DCE. This eliminates the frustrating and tedious task of logging into many different hosts to manage them. This also allows for scalability because it is impractical to manage a large system by logging into all its hosts.

The features of **dced** are greatly enhanced when used remotely. Of course, an administrator can use **dced** to locally manage a host's services, but **dced**'s real power is in remotely managing system and application server configurations, key tables, server startup, login configurations, and cell information.

Security becomes a major issue when it comes to remote services. With the power of **dced**'s services and **dcecp**, it is important that only authorized principals can use them. The **dced** program controls access to its various objects by using ACLs. Server keys are security-sensitive data that must be seldom transmitted over the network. All key table data is encrypted when it is transmitted for secure remote key table management.

Finally, the remote capabilities of **dced** give you real-time status of processes and services in DCE.

Managing a Host's Endpoint Map

Each DCE host has an endpoint map that contains a mapping of servers to endpoints. Each endpoint map server entry is associated with an array of services (interfaces) provided by the server, and each service is associated with an array of objects supported by the service.

When a typical server calls the **dce_server_register()** routine, the RPC runtime generates the endpoints on which the server will listen for calls and then uses **dced**'s endpoint mapper service of the local host to register the endpoints. Later, when a typical client makes a remote procedure call, its RPC runtime uses the server host’s endpoint mapper service to find the server. When the typical server shuts down, it calls the **dce_server_unregister()** routine to remove its endpoints from the endpoint map so that clients do not later try to bind to it.

Applications can also use the lower-level **rpc_ep_register()** and associated RPC routines. Because the endpoint map is essential for RPCs to work, endpoints are fully described in “Chapter 12. RPC Fundamentals” on page 169 and the endpoint map structure is described with respect to routing of RPCs in “Chapter 16. Writing Internationalized RPC Applications” on page 281.

The endpoint map is for the most part maintained automatically by **dced**. For example, it periodically removes stale endpoints so that the RPC runtime will not try to complete a binding for a client to a server that is no longer running. However, administrative applications may find it necessary to peruse a remote endpoint map and even remove specific endpoints from a local host’s endpoint map.

To read the elements of a remote endpoint map, applications use a loop with the set of routines **rpc_mgmt_ep_elt_inq_begin()**, **rpc_mgmt_ep_elt_inq_next()**, and **rpc_mgmt_ep_elt_inq_done()**. The inquiry can return all elements until the list is exhausted, or the inquiry can be restricted to return elements for the following:

- Elements matching an interface identifier (UUID and version number)
- Elements matching an object UUID
- Elements matching both an interface identifier and object UUID

Administrators can manage the endpoint map by using **dcecp** with the **endpoint** object.

You can use the **dced_server_disable_if()** routine to mark as disabled all the endpoints for a specific interface. This will prevent any new RPCs with partial bindings from binding to the server for this interface, but not prevent clients from using the interface if they already have a full binding with these endpoints. You can use the **dced_server_enable_if()** routine to reenable previously disabled interfaces. In an extreme situation, you could permanently remove endpoints directly from the local endpoint map by calling the **rpc_mgmt_ep_unregister()** routine. This function cannot be done remotely for security reasons.

Binding to dced’s Services

When you write a program that uses a host service, you begin by creating a **dced** binding to the service on a particular host. Bindings are relationships between clients and servers that allow them to communicate. A **dced** binding is a specific kind of binding that not only gives your application a binding to the **dced** [1] server but also associates the binding with a specific host service on that server.

In general, an application follows these basic steps to use a host service:

1. Establish a binding to the service on the desired host. For example, your application can establish a binding to the host data management service on another host.

1. Applications must establish a binding to each host service used. However, the endpoint mapper service uses a different binding mechanism and API from the other host services. This is due to the fact that the endpoint mapper service already existed within the very large RPC API in earlier versions of DCE, prior to the development of **dced**.

2. Obtain one or more **dced** entries for that service. For example, your application can obtain the **hostdata** entry that identifies the host's cell name, among other things. This step is valid for the following services:
   - hostdata management
   - server management
   - key table management

   Depending on the service and function desired, this step may or may not be necessary. For example, the security validation service does not store data, so **dced** maintains no entries for this service.
3. Access (read or write) the actual data for the entries obtained or perform other functions appropriate for the service. For example, if your application reads the hostdata management service's cell name entry, the API accesses **dced** which may actually read the data from a file. For another example, if your application established a binding to the security validation service, it could validate the security daemon.
4. Release the resources obtained in step 2.
5. Free the binding established in step 1.

Applications bind to a host service by using the **dced_binding_create()** or **dced_binding_from_rpc_binding()** routine. The first routine establishes a **dced** binding to a service on a host specified in a service name, and the second routine establishes a **dced** binding to a service on a host for which the application already has a binding. Both of the routines return a **dced** binding handle of type **dced_binding_handle_t**, which is used as an input parameter to all other **dced** API routines.

Host Service Naming in Applications

Applications include a host service name as input to the **dced** binding routine **dced_binding_create()**. A host service name is a string that may include a host name, or a cell and host name. The following key words in the host service name refer to a specific DCE host service:

hostdata
: Refers to configuration data of the hostdata management service.

srvrconf
: Refers to the static server configuration portion of the server management service. This refers to the management of a DCE-installed server.

srvrexec
: Refers to the dynamic server execution portion of the server management service. This refers to the management of a running DCE-installed server.

secval
: Refers to the security validation service.

keytab
: Refers to the private key data of the key table management service.

The following examples show service names and the locations of the hosts in the namespace:

service
: The host is local, the same as the application's.

service@**hosts/** *host*
: The host is in the local namespace.

/.:/hosts/ *host***/config/** *service*
: The complete specification for *service*@**hosts/** *host* where the host is in the local namespace.

/.../ *cell***/hosts/** *host***/config/** *service*
: The host is in the global namespace.

Because the **dced_binding_from_rpc_binding()** routine already knows which host to bind to from an RPC binding input parameter, it uses one of the global variables defined for each service (instead of a string) to specify which **dced** service to use.

The dced Program Maintains Entry Lists

One **dced** service's data is very different from another's (for example, server configuration data versus key table data), but you manipulate the data in a similar way. This is because it is a simpler and more efficient design to implement a few API routines that can handle more than one kind of data rather than many routines that do essentially the same thing but on a different service's data. An added benefit is a flexible API that can handle your own application's data and new kinds of DCE data in the future.

To separate the actual data from the API implementation, a **dced** service maintains a list of all data items in an *entry list*. Entry lists contain *entries* that describe the name and location of each item of data, but they do not contain the actual data. With this mechanism, **dced** can obtain and manipulate data very efficiently, without concern for the implementation and location of the actual data. It also supports well the model that administrators commonly need when accessing data: scan a list, select an item, and use the data.

The **dced** program maintains entry lists for the **hostdata**, **srvrconf**, **srvrexec**, and **keytab** services. The **secval** service does not need an entry list because it does not maintain any data, but functions are performed to set its state.

There is a special relationship between **srvrconf** and **srvrexec** entries. In order for **dced** to control the start of a server, the server must have a **srvrconf** entry associated with server configuration data. When **dced** starts a server, it generates from the **srvrconf** entry and data a **srvrexec** entry and associates the new entry with the running server's state.

Figure 1 on page 10 shows the entry lists maintained by **dced**.

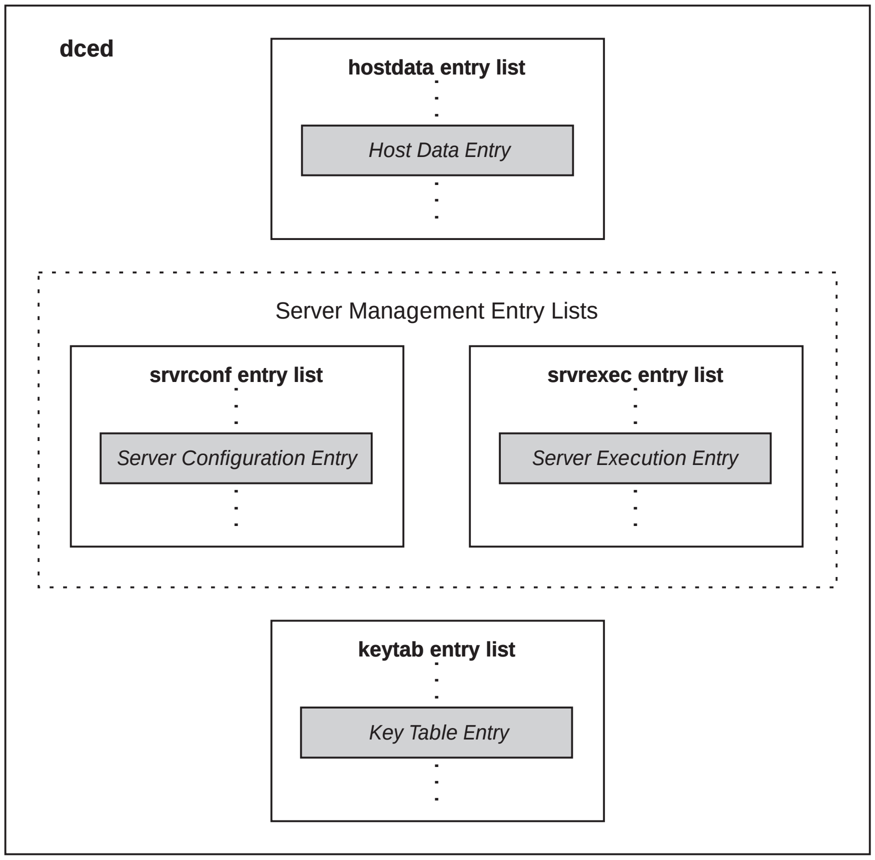


Figure 1. The dced Entry Lists

Although an entry can be associated with many different kinds of data items, all entries have the same structure, shown in Figure 2.

Entry UUID, Name, Description, Storage Tag

Figure 2. Structure of an Entry

Each entry is a **dced_entry_t** data structure. Each member of this data structure is described as follows:

id An entry UUID is necessary to uniquely identify the data item. Some data items have well-known UUIDs (the same UUID for the particular item on all hosts). The data type is **uuid_t**.

name Each data item is identified with a name, to which applications refer. The name need only be unique within an entry list because the entry UUID guarantees the entry's uniqueness. Some item names are well-known and defined in header files. The data type is **dced_string_t**.

description
This is a human-readable description of the data item. Its data type is **dced_string_t**.

storage_tag
The storage tag locates the actual data. Each service *knows* how to interpret this tag to find the data. For example, some data is stored in a file,

the name of which is contained in the storage tag. Other data is stored in memory and the storage tag contains a pointer to the memory location. The data type is **dced_string_t**.

Reading All of a Host Service's Data

Suppose you want to display host service data in an application that has a graphical user interface. The **dcecp** commands may not be adequate to display data for this application. The following example shows how to obtain the entire set of data for each host service:

```
dced_binding_handle_t  dced_bh;
dced_string_t      host_service;
void          *data_list;
unsigned32       count;
dced_service_type_t   service_type;
error_status_t     status;
 .
 .
 .
while(user_selects(&host_service, &service_type)){ /*application*/
                                                     /*specific */
  dced_binding_create(host_service,
     dced_c_binding_syntax_default,
         &dced_bh,
     &status);
  if(status == error_status_ok) {
    dced_object_read_all(dced_bh, &count, &data_list, &status);
    if(status == error_status_ok) {
      display(service_type, count, data_list);   /* application  */
                                                 /* specific  */
      dced_objects_release(dced_bh, count, data_list, &status);
    }
    dced_binding_free(dced_bh, &status);
  }
}
```

user_selects()
: This is an example of an application-specific routine that constructs the complete service name from host and service name information. Data is stored and retrievable for the **hostdata**, **srvrconf**, **srvrexecD**, and **keytab** services. No data is stored for the **secval** service.

dced_binding_create()
: Output from the **dced_binding_create** routine includes a **dced** binding handle whose data type is **dced_binding_handle_t**. If an application already has an RPC binding handle to a server on the host desired, it can use the **dced_binding_from_rpc_binding()** routine to bind to **dced** and one of its host services on that host. (Applications also use these routines to bind to the **secval** service to perform other functions.)

dced_object_read_all()
: Applications use the **dced_object_read_all()** routine to read data for all the objects in an entry list. The output includes the address of an allocated buffer of data and a count of the number of objects the buffer contains. The data type in the buffer depends on the service used.

display()
: This is an application-specific routine that displays the data. Before the data is displayed, it must be interpreted depending on the service. The **hostdata** data is an array of **sec_attr_t** data structures, the **srvrconf** and **srvrexec**

data are arrays of **server_t** structures, and the **keytab** data is an array of **dced_key_list_t** structures. The following code fragments show the data type for each service:

```
void display(
dced_service_type_t service_type, /* dced service type */
int         count,                /* count of the number of data items */
void        *data)                /* obtained from dced_object_read{_all}() */
{
  sec_attr_t          *host_data;
  server_t            *servers;
  dced_key_list_t     *keytab_data;
  .
  .
  .
  switch(service_type) {
  case dced_e_service_type_hostdata:
    host_data = (sec_attr_t *)data;
    . . .
  case dced_e_service_type_srvrconf:
    servers = (server_t *)data;
    . . .
  case dced_e_service_type_srvrexec:
    servers = (server_t *)data;
    . . .
  case dced_e_service_type_keytab:
    keytab_data = (dced_key_list_t *)data;
    . . .
  default:
    /* No other dced service types have data to read. */
    break;
  }
  return;
}
```

dced_objects_release()
Each call to the **dced_object_read_all()** routine requires a corresponding call to **dced_objects_release()** to release the resources allocated.

dced_binding_free()
Each call to the **dced_binding_create()** routine requires a corresponding call to **dced_binding_free()** to release the resources for the binding allocated.

Managing Individual dced Entries

Figure 3 on page 13 shows examples of individual **dced** entries and the locations of associated data. The data item name or its UUID is used to find an entry, and then the storage tag is used to find the data.

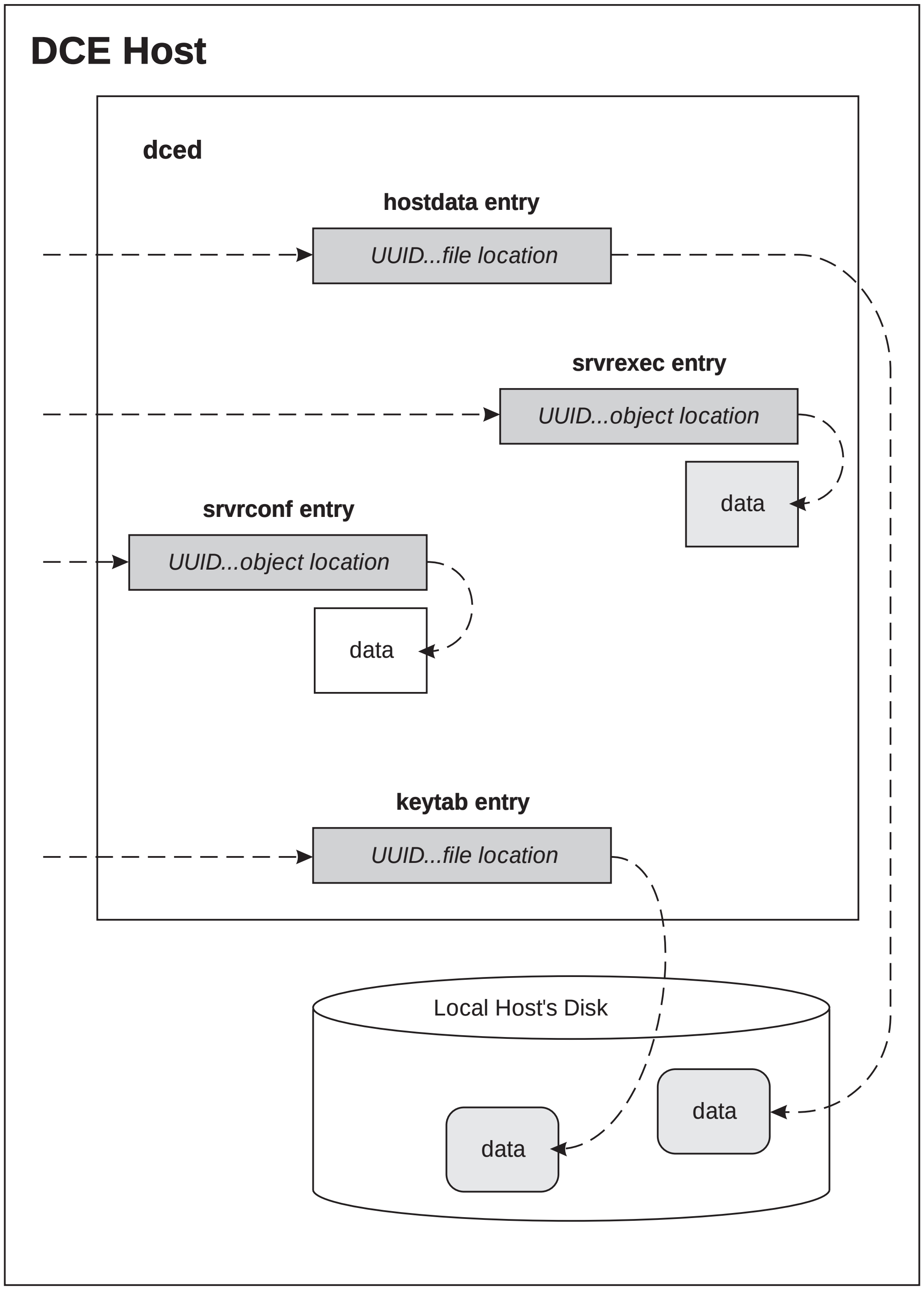


Figure 3. Accessing Hostdata

The data for each **hostdata** item is stored in a file on disk. The **dced** program uses the UUID to find the entry in the **hostdata** entry list. The entry's storage tag is then used to find the data. For **hostdata**, the tag contains a filename in OSF's reference implementation. The data returned for one entry is an array of strings in a **sec_attr_t** structure.

The server management data is stored in memory. The **dced** program uses UUIDs (maintained in the entry lists by **dced**) to find an entry. The location of the data in memory is indicated by the storage tag. The data returned for one entry is a structure of server data (**server_t**). All data for the **srvrconf** and **srvrexec** entries are accessed from memory for fast retrieval, but the **srvrconf** data is also stored on disk for use when a host needs to reboot.

Each **keytab** entry stores its data in a file on disk. However, like the server management entries, the **keytab** entries use server names and corresponding UUIDs (maintained by **dced**) to identify each entry. The storage tag contains the name of the key table file. The data returned for one entry is a list of keys of type **dced_key_list_t**.

The following example shows how to obtain and manage individual entries for the **hostdata**, **srvrconf**, **srvrexec**, or **keytab** services:

```
handle_t        rpc_bh;
dced_binding_handle_t  dced_bh;
dced_entry_list_t    entries;
unsigned32       i;
dced_service_type_t   service_type;
void          *data;
error_status_t     status;
 .
 .
 .
dced_binding_from_rpc_binding(service_type, rpc_bh, &dced_bh, &status);
if(status != error_status_ok)
  return;
dced_list_get(dced_bh, &entries, &status);
if(status == error_status_ok) {
  for(i=0; i<entries.count; i++) {
    if(select_entry(entries.list[i].name)) {/* application specific */
      dced_object_read(dced_bh, &(entries.list[i].id), &data, &status);
      if(status == error_status_ok) {
        display(service_type, 1, &data); /* application specific */
        dced_objects_release(dced_bh, 1, data, &status);
      }
    }
  }
  dced_list_release(dced_bh, &entries, &status);
}
dced_binding_free(dced_bh, &status);
```

Each routine is described as follows:

dced_binding_from_rpc_binding()

The **dced_binding_from_rpc_binding()** routine returns a **dced** binding handle whose data type is **dced_binding_handle_t**. This binding handle is used in all subsequent **dced** API routines to access the service. The host is determined from the RPC binding handle, *rpc_bh*, and the *service_type* is selected from the following list:

- **dced_e_service_type_hostdata**
- **dced_e_service_type_srvrconf**
- **dced_e_service_type_srvrexec**
- **dced_e_service_type_keytab**

dced_list_get()

Applications use the **dced_list_get()** routine to get a service's entire list of names. Using the **dced_list_get()** routine gives your application great flexibility when manipulating entries in an entry list. If you prefer, your application can use the **dced_entry_cursor_initialize()**,

dced_entry_get_next(), and **dced_entry_cursor_release()** set of routines to obtain individual entries, one at a time.

select_entry()
: This is an application-specific routine that selects which entry to use based on the entry name.

dced_object_read()
: The default attribute for **dced_object_read()** is to return an array of strings. The **hostdata** and **keytab** services have other read routines that allow you to specify binary data.

display()
: This is an example of an application-specific routine that simply displays the server configuration data read. Depending on the service, a different data structure is used. For the **hostdata** service, a **sec_attr_t** is used. For the **srvrconf** and **srvrexec** services **server_t** structures are used. For the **keytab** service, a **dced_key_list_t** structure is used.

dced_objects_release()
: After your application is finished with the data read with the **dced_object_read()** routine, free the buffer of allocated data by using the **dced_objects_release()** routine.

dced_list_release()
: Each call to the **dced_list_get()** routine requires a corresponding call to **dced_list_release()** to release the resources allocated for the entry list.

dced_binding_free()
: Each call to the **dced_binding_from_rpc_binding()** routine requires a corresponding call to **dced_binding_free()** to release the resources of the allocated binding.

Managing Hostdata on a Remote Host

Administrators typically use the **dcecp hostdata** object to remotely manage the data of the **hostdata** service. However, application developers can use the **dced** API for their own management applications or if **dcecp** does not handle a task in the desired way, such as for a browser of hostdata that uses a graphical user interface.

Kinds of Hostdata Stored

Each **hostdata** item is stored in a file, and **dced** has a UUID associated with each. The standard data items include the following well-known names:

cell_name
: The name of the cell to which your host belongs is stored.

cell_aliases
: When the cell name changes, the old names are designated as cell aliases.

dce_cf.db
: The DCE configuration data file is stored.

host_name
: The host name is stored.

pe_site
: The location of the security server is stored.

post_processors

The **post_processors** file contains UUID-program pairs for which the UUIDs represent other **hostdata** items. If changes occur to an associated **hostdata** item, the system runs the program.

svc_routing

The default routing file for serviceability messages is stored.

Depending on your DCE provider, additional items may exist. In addition to the well-known **hostdata** items, applications can also add their own. The well-known **hostdata** items have well-known UUIDs defined in the file **/usr/include/dce/dced_data.h**, but you can use the **dced_inq_uuid()** routine to obtain any UUID associated with any name known to **dced**.

Adding New Hostdata

In addition to modifying existing hostdata, you can add your own data by using the hostdata API. For example, suppose you want to add a printer to a host and make the configuration file part of that host's **dced** data. The following example shows how to do this:

```
dced_binding_handle_t  dced_bh;
error_status_t      status;
dced_entry_t       entry;
dced_attr_list_t     data;
int           num_attr, str_size;
sec_attr_enc_str_array_t *attr_array;
 .
 .
 .
dced_binding_create(dced_c_service_hostdata,
       dced_c_binding_syntax_default,
     &dced_bh,
   &status);
/*Create Entry Data */
uuid_create(&(entry.id), &status);
entry.name = (dced_string_t)("NEWERprinter");
entry.description = (dced_string_t)("Configuration for a new printer.");
entry.storage_tag = (dced_string_t)("/etc/NEWprinter");
/* Create the Attributes, one for this example */
data.count = 1;
num_attr = 1;
data.list = (sec_attr_t *)malloc(data.count * sizeof(sec_attr_t) );
(data.list)->attr_id = dced_g_uuid_fileattr;
(data.list)->attr_value.attr_encoding = sec_attr_enc_printstring_array;
str_size = sizeof(sec_attr_enc_str_array_t) +
           num_attr * sizeof(sec_attr_enc_printstring_p_t);
attr_array = (sec_attr_enc_str_array_t *)malloc(str_size);
(data.list)->attr_value.tagged_union.string_array = attr_array;
attr_array->num_strings = num_attr;
attr_array->strings[0] = (dced_string_t)("New printer configuration data");

dced_hostdata_create(dced_bh, &entry, &data, &status);
dced_binding_free(dced_bh, &status);
```

The description of this example is as follows:

dced_binding_create()

This routine creates a **dced** binding to a **dced** service. The binding handle created is used in all subsequent calls to appropriate **dced** API routines. By using the **dced_c_server_hostdata** value for the first parameter, we are using the **hostdata** service on the local host.

Create Entry Data
Prior to creating a **hostdata** entry, we have to set its values. These include the name and UUID that **dced** will use to identify the new data, a description of the entry, and a filename with the full pathname of where the actual data will reside.

Create the Attributes
The data stored is of type **sec_attr_t**. This data type is a very flexible one that can store many different kinds of data. In this example, we set the file to have one attribute, printable string information. This example has only one string of data. You can also establish binary data for the file.

dced_hostdata_create()
This routine takes the binding handle, entry, and new data as input; it creates the file with the new data and returns a status code.

If the printer configuration file already exists on the host, but you want to now make it accessible to **dced**, use the **dce_entry_add()** routine instead of **dced_hostdata_create()**.

dced_binding_free()
Each call to the **dced_binding_create()** routine requires a corresponding call to **dced_binding_free()** to release the binding resources allocated.

Use the **dced_hostdata_delete()** routine to delete application-specific **hostdata** items and their entries. For example, the printer installed in the example is easily removed with this routine. If you are only taking the printer out of service for a short time, use the **dced_entry_remove()** routine to remove the **dced** entry but not the data file itself. When the printer is later ready again, use the **dced_entry_add()** routine to reinstall it.

Do not delete the well-known **hostdata** items or remove their entries.

Modifying Hostdata

Changing hostdata cannot only change the way the host works but it also affects other files and processes on the host. Therefore, care should be taken when changing hostdata. Deleting the well-known **hostdata** entries can cause even more serious operational problems for the host.

The current as well as earlier versions of DCE provide configuration routines that use a **dce_cf.db** file for data. When hostdata changes, **dced** also makes the appropriate changes to this file so that the **dce_cf*** routines continue to work correctly. This is one reason the **hostdata** items are established as well-known names with well-known UUIDs so that **dced** knows which values to monitor.

Management applications use the **dced_hostdata_read()** routine to obtain the data for an entry referred to by an entry UUID. To modify an entry's actual data, applications use the **dced_hostdata_write()** routine. This routine replaces the old data with the new data for the **hostdata** entry represented by the entry UUID. The **hostdata** entry must already exist because this routine will not create it. Use the **dced_hostdata_create()** routine to create new **hostdata** entries.

Running Programs Automatically When Hostdata Changes

The following example shows how to use the **post_processors** feature of the well-known hostdata to cause **dced** to automatically run a program if another

hostdata entry changes. In this example, the **post_processors** file is read, and data is added for the **NEWERprinter hostdata** entry created in an earlier example. The data is placed in a **dced_attr_list_t** structure and written back to the **post_processors hostdata** entry.

```
dced_binding_handle_t dced_bh;
uuid_t        entry_uuid;
sec_attr_t      *data_ptr;
error_status_t    status;
int          i, num_strings, str_size;
sec_attr_enc_str_array_t *attr_array;
unsigned_char_t    *string_uuid, temp_string[200];
dced_attr_list_t   attr_list;

dced_binding_create(dced_c_service_hostdata,
       dced_c_binding_syntax_default,
       &dced_bh,
   &status);
dced_hostdata_read(dced_bh,
          &dced_g_uuid_hostdata_post_proc,
          &dced_g_uuid_fileattr,
          &data_ptr,
          &status);

/* Create New Array and Copy Old Data into it */
num_strings = data_ptr->attr_value.tagged_union.string_array->num_strings + 1;
str_size = sizeof(sec_attr_enc_str_array_t) +
           num_strings * sizeof(sec_attr_enc_printstring_p_t);
attr_array = (sec_attr_enc_str_array_t *)malloc(str_size);
attr_array->num_strings = num_strings;
for(i=0; i<(num_strings-1); i++) {
  attr_array->strings[i] =
    data_ptr->attr_value.tagged_union.string_array->strings[i];
}

dced_inq_id(dced_bh, "NEWERprinter", &entry_uuid, &status);

uuid_to_string(&entry_uuid, &string_uuid, &status);
sprintf(temp_string, "%s %s", string_uuid, "/path/and/program/to/run");
attr_array->strings[num_strings-1] = (dced_string_t)(temp_string);
data_ptr->attr_value.tagged_union.string_array = attr_array;
attr_list.count = 1;
attr_list.list = (sec_attr_t *)malloc(attr_list.count * sizeof(sec_attr_t));
attr_list.list = data_ptr;
dced_hostdata_write(dced_bh,
          &dced_g_uuid_hostdata_post_proc,
          &attr_list,
          &status);

dced_objects_release(dced_bh, 1, (void*)(data_ptr), &status);
dced_binding_free(dced_bh, &status);
```

The description of this example is as follows:

dced_binding_create()
This routine creates a **dced** binding to the **hostdata** service on a specified host. The binding handle created is used in all subsequent calls to appropriate **dced** API routines. The **dced_c_service_hostdata** argument is a constant string that is the well-known name of the **hostdata** service. When this string is used by itself, it refers to the service on the local host.

dced_hostdata_read()
This routine reads the **hostdata** item referred to by the entry UUID. In this example, the global variable **dced_g_uuid_hostdata_post_proc** represents the UUID for the well-known **post_processors** file. The second

parameter specifies an attribute for the data. Attributes describe how the data is to be interpreted. In this example, we know the data to be read is plain text, so we use the global variable **dced_g_uuid_fileattr** to specify plain text rather than binary data (**dced_g_uuid_binfileattr**).

Create a New Array
The next few lines copy the existing array of print strings into a new array that has additional space allocated for the new data.

dced_inq_id()
This routine acquires the UUID **dced** that maintains for a known entry name. In this example, we need the UUID for the **NEWERprinter hostdata** entry so that it can be included in the data stored back in the **post_processors** file.

uuid_to_string()
This routine returns the string representation of a UUID. Each line in the **post_processors** file contains a string UUID and a program name for **dced** to run if the **hostdata** entry referred to by the UUID changes. The next few lines create a new string containing the string UUID and a program name, adds the new string to the new array, and reassigns the new array to the old data pointer.

dced_hostdata_write()
Since hostdata could have more than one attribute associated with each entry, the data must be inserted in an attribute list data structure before the **dced_hostdata_write()** routine is called. In the case of the well-known **post_processors hostdata** object, the attribute is for a plain text file. The **dced_hostdata_write()** routine replaces the old data with the new data for the **hostdata** entry represented by the entry UUID.

dced_objects_release()
Each call to the **dced_hostdata_read()** routine requires a corresponding call to **dced_objects_release()** to release the resources allocated.

dced_binding_free()
Each call to the **dced_binding_create()** routine requires a corresponding call to **dced_binding_free()** to release the resources allocated.

The **post_processors** data for this **dced** now contains an additional string with a UUID and program name. If the **hostdata** item represented by the UUID for **NEWERprinter** is changed, **dced** automatically runs the program.

Controlling Servers Remotely

Both applications developers and system administrators may want servers to have certain support services and control functionality. For example, servers may need mechanisms to store operational data, and they may need to start or stop in various ways. The **dced** program provides these support and control mechanisms for servers.

Servers are typically configured by an administrator using the **dcecp server** object in a script after the server is installed on the host. In addition to configuring the server, this script would commonly include other tasks like create an account and assign a principal name for the server, modify the ACLs and key table files (keytabs) to control access to the server and its resources, and export the server binding information to the Cell Directory Service (CDS) so that clients can find a server that will start dynamically later.

After a server is configured, whether it runs as a persistent daemon or an on-demand (dynamic) process, administrators would again use **dcecp** if they need to control or modify its behavior. Although server management is typically an administrator's task, you may want a management application to perform these tasks, including the following:

- Configure a server to describe how it can be invoked
- Start a server based on configuration data
- Stop a running server
- Disable a specific service provided by a running server
- Enable a specific service for a running server
- Modify a server's configuration
- Delete a server's configuration, effectively removing the server from **dced**'s control

Two States of Server Management: Configuration and Execution

If all servers ran as persistent processes, **dced** could maintain data about each server in a single (albeit complex) data structure. However, due to the fact that some servers may run on demand, it is a more flexible design to have two sets of data: one that describes the default configuration to start the server, and one that describes the executing (running) server. Earlier in this chapter when we described **dced** service naming, we defined **srvrconf** and **srvrexec** objects to name the two portions of the server management service.

Table 1 lists the routines that applications can use to control servers. It also shows the valid object names to use when establishing a **dced** binding prior to using the routine.

Table 1. API Routines for Remote Server Management

API Routine	Service Name for Binding
dced_server_create()	**srvrconf**
dced_server_start()	**srvrconf**
dced_server_disable_if()	**srvrexec**
dced_server_enable_if()	**srvrexec**
dced_server_stop()	**srvrexec**
dced_object_read()	**srvrexec**or **srvrconf**
dced_object_read_all()	**srvrexec**or **srvrconf**
dced_server_modify_attributes()	**srvrconf**
dced_server_delete()	**srvrconf**

Configuring Servers

Although administrators commonly use **dcecp** to configure servers remotely, management applications can use **dced** API routines to configure a new server remotely by creating server configuration data, changing a remote server's configuration, and deleting a server's configuration data.

Configuring a New DCE Server

Management applications use the **dced_server_create()** routine to add a new server to a host. After a server is configured, it can be remotely controlled by modifying its configuration attributes, starting and stopping it, enabling or disabling the RPC interfaces it supports, and deleting its configuration.

Configuring the server involves describing the server for DCE by allocating and filling in a **server_t** data structure, as shown in the following example. Note that not all **server_t** fields are assigned values in this example.

```
int          i;
dced_binding_handle_t dced_bh;
server_t       conf, exec;
dced_string_t     server_name;
uuid_t       srvrconf_id, srvrexec_id;
dced_attr_list_t   attr_list;
error_status_t    status;
static service_t   nil_service;
 .
 .
 .
dced_binding_create("srvrconf@hosts/somehost",
          dced_c_binding_syntax_default,
          &dced_bh,
          &status);
dced_inq_id(dced_bh, server_name, &srvrconf_id, &status);
if(status == error_status_ok) {
  puts("Configuration already exists for this server.");
  dced_binding_free(dced_bh, &status);
  return;
}
/* ___________setup a server_t structure ________________*/
uuid_create(&(conf.id), &status);
conf.name      = server_name;
conf.entryname   = (dced_string_t)"/.:/greeter";
conf.services.count = 1;
/* ___service_t structures represent each interface supported ___*/
conf.services.list =
  (service_t *)malloc(conf.services.count * sizeof(service_t));
for(i=0; i<conf.services.count; i++) {
  rpc_if_inq_id(greetif_v1_0_c_ifspec,
         &(conf.services.list[i].ifspec),
         &status);
  conf.services.list[i] = nil_service;
  conf.services.list[i].ifname   = (dced_string_t)"greet";
  conf.services.list[i].annotation = (dced_string_t)"The greet application";
  conf.services.list[i].flags   = 0;
}

/* ______________server_fixedattr_t structure _______________*/
conf.fixed.startupflags =
  server_c_startup_explicit | server_c_startup_on_failure;
conf.fixed.flags = 0;
conf.fixed.program = (dced_string_t)"/server/path/and/program/name";

dced_server_create(dced_bh, &conf, &status);
dced_binding_free(dced_bh, &status);
```

dced_binding_create()
: To configure a server, the application must first create a **dced** binding to the **srvrconf** portion of the server management service on a specified host. The binding handle created is used in all subsequent calls to appropriate dced API routines.

dced_inq_id()
: This routine returns the UUID that **dced** associates with the name input.

Each configured server has an associated UUID used by **dced** to identify it. In this example, we won't try to create a configuration for a server that already exists.

Set Up a server_t Structure for the Server
The **server_t** structure contains all the information DCE uses to specify a server.

Set Up service_t Structures for Each Interface
Each service that the server supports is represented by a **service_t** data structure that contains the interface specification, among other things. In this example the client stub for the interface was compiled with the program so that the interface specification (**greetif_v1_0_c_ifspec**) could be obtained without building the structure from scratch.

Set Up a server_fixedattr_t Structure
Other fixed attributes required for all servers describe how the server can start, the program name and pathname for the server so that **dced** knows which program to start, and the program's arguments, among other things.

dced_server_create()
This routine uses the filled-in **server_t** structure to create a **srvrconf** entry for **dced**. The data is stored in memory for quick access whenever the server is started.

dced_binding_free()
Each call to the **dced_binding_create()** routine requires a corresponding call to **dced_binding_free()** to release the binding resources allocated.

Modifying a Server's Configuration Attributes

The data for configuring servers includes arrays of attributes. For flexibility, **dced** is implemented using the extensible and dynamic data structures developed for the DCE security registry attributes. This extended registry attribute (ERA) schema gives vendors the flexibility to modify the attributes appropriate for configuring servers on various systems. The use and modification of these data structures are described in "Chapter 28. The Registry API" on page 545.

Applications commonly use **dced_server_modify_attributes()** after the **dced_server_create()** routine to change the default configuration attributes (the **attributes** field of a **server_t** structure) for a remote server. A **dced_attr_list_t** data structure is input that contains an array of **sec_attr_t** data structures and a count of the number in the array.

Deleting a DCE Server

Management applications use **dced_server_delete()** to delete a server's configuration data and entry in its hosts **dced**. Although this does not delete the actual server program from the host, it removes it from DCE control.

Starting and Stopping Servers

A server typically runs as persistent process or is started on demand when a client makes a remote procedure call to it. Management applications can start remote servers by using the **dced_server_start()** routine. This is a **srvrconf** routine that takes as input server configuration data in the form of an attribute list.

Once a server has started, it tends to remain running until an administrator or management application stops it, but some applications may stop themselves if, for example, they do not detect activity within a specified time. To stop remote servers, applications can use the **dced_server_stop()** routine.

The following example shows how an application starts or stops a server:

```
dced_binding_handle_t dced_bh, conf_bh, exec_bh;
server_t       conf, exec;
dced_string_t     server_name;
uuid_t       srvrconf_id, srvrexec_id;
error_status_t    status;
 .
 .
 .
/* Toggle the Starting or Stopping of a Server */
dced_binding_create("srvrconf@hosts/somehost",
          dced_c_binding_syntax_default,
          &conf_bh,
          &status);
dced_binding_create("srvrexec@hosts/somehost",
          dced_c_binding_syntax_default,
          &exec_bh,
          &status);
dced_inq_id(exec_bh, server_name, &srvrexec_id, &status);
if(status != error_status_ok) {
  puts("Server is NOT running.");
  dced_inq_id(conf_bh, server_name, &srvrconf_id, &status);
  dced_server_start(conf_bh, &srvrconf_id, NULL, &srvrexec_id, &status);
}
else {
  puts("Server is RUNNING.");
  dced_server_stop(exec_bh, &srvrexec_id, srvrexec_stop_rpc, &status);
}
dced_binding_free(conf_bh, &status);
dced_binding_free(exec_bh, &status);
```

dced_binding_create()
These routines create **dced** bindings to the **srvrconf** and **srvrexec** portions of the server management service on a specified host. The binding handles created are used in all subsequent calls to appropriate **dced** API routines.

dced_inq_id()
This routine returns the UUID that **dced** associates with the name input. Each name used to identify an object of each service has a UUID. If **dced** maintains a UUID for a **srvrexec** object, the server is running. However, it is possible that the server is in an in-between state as it is starting up or shutting down. For a more robust check as to whether the server is running, use the **dced_object_read()** routine to read the **server_t** structure for the **srvrexec** object. If the *exec_data.tagged_union.running_data.instance* UUID is the same as the **srvrconf** UUID (*srvrconf_id*), the server is running.

dced_server_start()
This routine starts the server via **dced**. The **srvrconf** binding handle and UUID are input. For special server configurations, you can start a server with a specific list of attributes, but a value of NULL in the third parameter uses the attributes of the server configuration data. You can input a **srvrexec** UUID for **dced** to use, or allow it to generate one for you.

dced_server_stop()
This routine stops a running server identified by its **srvrexec** UUID. The cleanest stop method is to cause **dced** to use the

rpc_mgmt_server_stop_listening() routine so that all outstanding remote procedure calls complete before the server stops.

dced_binding_free()
Each call to the **dced_binding_create()** routine requires a corresponding call to **dced_binding_free()** to release the binding resources allocated.

Enabling and Disabling Services of a Server

Most servers have all their services enabled to process all requests. However, a server may need to enable or disable services to synchronize them, for example. For another example, an administrator (or management application) may need to disable or enable services to perform orderly startup or shutdown of a server.

Each service provided by a server is implemented as a set of procedures. DCE uses an interface definition to define a service and its procedures, and application code refers to the interface when controlling the service.

When a server starts, it initializes itself by registering with the RPC runtime and the **dced** process on its host by using the **dce_server_register()** routine. This enables all services (interfaces) that the server can support. The server can then disable and reenable services (in whatever order it requires) by using the **dce_server_disable_if()** and **dce_server_enable_if()** routines.

To control the services of remote servers, management applications use the **dced_server_disable_if()** and **dced_server_enable_if()** routines. These routines work on the **srvrexec** object. When a service (interface) is disabled, a client that already knows about the service (through a binding handle to this interface and server) will no longer work because the interface is unregistered with the RPC runtime. If you wish to have clients that already know about the server and service work, but wish to prohibit any new clients from finding the server and service, you can use **rpc_mgmt_ep_unregister()** to remove from the endpoint map the server address information with respect to the service. This routine does not affect the RPC runtime.

Validating the Security Server

The security validation service (**secval**) has the following major functions:

- It maintains a login context for the host's self-identity which includes periodic changes to the host's key (password).
- It validates and certifies to applications, usually login programs, that the DCE security daemon (**secd**) is legitimate.

Clients (including remote clients, local servers, host logins, and administrators) all need the security validation service to make sure that the **secd**) process being used by the host is legitimate. The security validation service establishes the link in a trust chain between applications and **secd** so that applications can trust the DCE security mechanism.

An application can trust its host's security validation service because they are on the same host, but an application has no way to convince itself that **secd**, presumably on another host, is genuine. However, if the application trusts another principal (in this case, the security validation service), which in turn trusts **secd**, then the trust chain now extends from the application to **secd**.

Typically, a login program accesses the security validation service when it uses the DCE Security Service's login API, described in "Chapter 29. The Extended Attribute API" on page 551. Administrators access the **secval** service by using the **dcecp secval** object. However, suppose you are writing a security monitoring application to watch for and respond to security attacks. After the application binds to the **secval** service, it can call the **dced_secval_validate()** routine to verify that the **secd** process is legitimate.

Applications can also use the **dced_secval_start()** and **dced_secval_stop()** routines to start and stop the security validation service on a given host.

For example, during configuration of a host, the **dced** program can start with or without the security validation service. Later when security is configured, a management application can start **secval** by using the **dced_secval_start()** routine. For another example, suppose our security monitoring application mentioned earlier suspects an attack. The application can call **dced_secval_stop()** to stop the security validation service without stopping the entire **dced**. This makes the login environment more restrictive.

Managing Server Key Tables

Keys for servers are analogous to passwords for human users. Keys also play a major role in authenticated remote procedure calls. Keys have some similarities with passwords. For example, server keys and user passwords have to follow the same change policy (or a more stringent one) for a given host or cell. This means that, just as a user has to periodically come up with a new password, a server has to periodically generate a new key. It is easy to see that a human user protects a password by memorizing it. But a server memorizes a key by storing it in a file called a key table.

It is more complex for a server to change keys than it is for a human user to change a password. For example, a human user needs to only remember the latest password, but a server may need to maintain a history of its keys by using version numbers so that currently active clients do not have difficulty completing a remote procedure call. When a client prepares to make authenticated remote procedure calls, it obtains a ticket to talk with the server. (The security registry of the authentication service encrypts this ticket by using the server's key, and later the server decrypts the ticket when it receives the remote procedure call.)

Timing can become an issue when a client makes a remote procedure call because tickets have a limited lifetime before they expire, and servers must also change their keys on a regular basis. Assuming the client possesses a valid ticket, suppose that, by the time the client makes the call, the server has generated a new key. If a server maintains versions of its keys, the client can still complete the call. Authentication is described in detail in "Chapter 23. Overview of Security" on page 483.

A key table usually contains keys stored by one server, and it must be located on the same host as that server. However, a key table can hold keys for a set of related servers, as long as all the servers reside on the same host. Servers usually maintain their own keys, and "Chapter 30. The Login API" on page 581 describes the API they use. Administrators can remotely manage key tables and the keys in the tables by using the **dcecp keytab** object. This section describes the API routines that management applications can use to manage the key tables and keys of other servers on the network.

Suppose you discover that a server or an entire host's security has been compromised. Applications can use the **dced_keytab_change_key()** routine to change a key table's key. The following example shows how to reset the key for all key tables on a specified host:

```
dced_binding_handle_t  dced_bh;
dced_entry_list_t    entries;
unsigned32       i;
error_status_t     status;
dced_key_t       key;
dced_binding_create("keytab@hosts/somehost",
           dced_c_binding_syntax_default,
           &dced_bh,
           &status);
dced_binding_set_auth_info(dced_bh,
              rpc_c_protect_level_default,
              rpc_c_authn_default,
              NULL,
              rpc_c_authz_dce,
              &status);

dced_list_get(dced_bh, &entries, &status);

for(i=0; i<entries.count; i++) {
  generate_new_key(&key); /* application specific */
  dced_keytab_change_key(dced_bh, &entries.list[i].id, &key, &status);
}
dced_list_release(dced_bh, &entries, &status);
dced_binding_free(dced_bh, &status);
```

dced_binding_create()
This routine creates a **dced** binding to a **dced** service on a specified host. The binding handle created is used in all subsequent calls to appropriate **dced** API routines. The **keytab** portion of the first argument represents the well-known name of the keytab service. When this string is used by itself, it refers to the service on the local host.

dced_binding_set_auth_info()
Accessing keytab data requires authenticated remote procedure calls. The **dced_binding_set_auth_info()** routine sets authentication for the **dced** binding handle, *dced_bh*.

dced_list_get()
Applications use the **dced_list_get()** routine to get a service's entire list of names.

generate_new_key()
This application-specific routine generates the new key and fills in a **dced_key_t** data structure. This routine could use the **sec_key_mgmt_gen_rand_key()** routine to randomly generate a new key.

dced_keytab_change_key()
The **dced_keytab_change_key()** routine tries to change the principal's key in the security service's registry first. If that is successful, it changes the key in the key table.

dced_list_release()
Each call to the **dced_list_get()** routine requires a corresponding call to **dced_list_release()** to release the resources allocated for the entry list.

dced_binding_free()
Each call to the **dced_binding_create()** routine requires a corresponding call to **dced_binding_free()** to release the resources allocated for a **dced** binding handle.

For more detailed key table management, applications can peruse a key table's list of keys by using the **dced_keytab_initialize_cursor()**, **dced_keytab_get_next_key()**, and **dced_keytab_release_cursor()** routines. Reading key table data remotely presents a greater security risk because data is sent over the network. For remote access, these routines actually get all the keys during one remote procedure call to be more efficient and to minimize the time keys are being sent over the network.

Earlier in this section we described how to change the key of a key table with the **dced_keytab_change_key()** routine. The key table management service also provides the routines **dced_keytab_add_key()** and **dced_keytab_remove_key()** to control key modification in even greater detail.

Finally, you can create a new key table by using **dced_keytab_create()**, or you can delete an existing key table by using **dced_keytab_delete()**.

Sample dced Application

The following sections contain the complete source code, Makefile, and **dcecp** installation scripts for a simple DCE application that uses some of the **dced** server management facilities.

The **greet_dced** application is an adaptation of the **greet** application described in the *Introduction to OSF DCE*. The **greet_dced** server is registered and executed via **dced**.

Once started, **greet_dced**'s behavior is identical to that of **greet**. 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 back to the client. The client prints the server's reply and terminates. (Note that the server does not catch signals, so when it is stopped it does not clean up its namespace or registry entries; this must be done manually.)

Running the Program

To run **greet_dced**, do the following:

1. Build the program by invoking the **make** command.
2. Change the **uid** and **gid** values in **greet_dced.install** according to your preferences. If you do change them, make sure that you chown the **keytab** file to the same **uid** in Step 4 below.
3. As **cell_admin**, do:
   ```
   dcecp greet_dced.install
   ```
   This creates a server principal and account with the password "secret", creates a CDS directory and changes permissions on it (so that the server principal has rights to create its server entry), creates a **keytab** entry and creates a **srvrconf** entry. It handles errors, so if something fails (e.g. if the user already exists) the program will still run to completion.
4. As root, do:

```
chown your_user_name greet_dced.ktab
```

This is necessary because the form of the **chown** command in **greet_dced.install** will fail—it is present there only as a reminder. If you use a different **uid** in the script, change it here as well.

5. As **cell_admin**, do:

   ```
   dcecp -c server start greeter_dced
   ```

6. Wait a few moments and check **/tmp/srv.out** to make sure the server has started.
7. Start the client as follows:

   ```
   ./greet_dced_client /.:/subsys/my_company/greet_dced/greeter_dced_entry
   ```

After you are done, you can get rid of everything as follows:

1. As **cell_admin**, stop the server:

   ```
   dcecp -c server stop greeter_dced -method soft
   ```

2. As **cell_admin**, run the delete script:

   ```
   dcecp greet_dced.delete
   ```

The program has the following limitations:

1. The server does not catch signals, so when it is stopped it does not clean up anything.
2. The **dce_server_sec_begin()** routine logs in using the server principal and keytab specified in the **srvrconf** file. It also starts a thread to manage the server's key. However, it does *not* start a thread to refresh the server's login context. That still needs to be done by the application writer, using the same method that was used in DCE 1.0.x:

   ```
   create a thread to run the following:

   loop
   find out when the login context expires
   do a pthread_delay_np for
   (expiration time - current time - 10 minutes)
   sec_login_refresh_identity();
   sec_key_mgmt_get_key();
   sec_login_validate_identity();
   sec_key_mgmt_free_key();
   end loop
   ```

greet_dced.idl

Following are the contents of the **greet_dced.idl** file.

```
/*
 * greet_dced.idl
 *
 * The "greet_dced" interface.
 */

[uuid(3d6ead56-06e3-11ca-8dd1-826901beabcd),
version(1.0)]

interface greet_dcedif
{
  const long int REPLY_SIZE = 100;

  void greet_dced(
```

```
    [in]      handle_t h,
    [in, string]  char client_greeting[],
    [out, string]  char server_reply[REPLY_SIZE]
  );
}
```

greet_dced_server.c

Following are the contents of the **greet_dced_server.c** file, which contains the **greet_dced** server setup and cleanup routines. This is where the server's interaction with **dced** takes place.

```
/* greet_dced_server_dce.c
 * Main program (initialization) for "greet_dced" server.
 * NEW SERVER for DCE 1.1.
 */

#include <stdio.h>
#include <dce/dced.h>
#include "greet_dced.h"
#include "util.h"

char invocation_instructions[] = "Usage:\n\
1. Invoke the dcecp program: dcecp\n\
  dcecp>\n\
2. Start the server:\n\
  dcecp> server start greeter_dced\n\
3. If dced cannot find a matching server object,
  create server configuration:\n\
  dcecp> source greet_dced.install\n\
  dcecp> server start greeter_dced\n\
4. exit dcecp.\n\
  dcecp> exit";

FILE * out = NULL;

boolean32 rpc_mgmt_authorize (rpc_binding_handle_t client_bn,
   unsigned32 op_no,
   unsigned32 *st);

int main(
  int argc,
  char *argv[]
)
{
  server_t          *server_conf;
  dce_server_register_data_t register_data[1];
  dce_server_handle_t     server_handle;
  error_status_t       status;

  /* if we are a daemon stderr is missing */
  out = fopen ("/tmp/srv.out" , "w");

  /* otherwise just use
  out = stderr;
  */

  fprintf(out, "Server start\n");  fflush(out);

  /********** Get the server's configuration from the local dced ******/
  fprintf(out, "dce_server_inq_server() call\n");
  fflush(out);
  dce_server_inq_server(&server_conf, &status);
  fprintf(out, "dce_server_inq_server() return\n");
  fflush(out);
  if(status != error_status_ok) { /* Describe startup via dcecp */
                  /* and dced */
    fprintf(out, "%s\n", invocation_instructions);
    fflush(out);
    ERROR_CHECK(status, "Cannot get server configuration structure");
  }

  /********** login and manage key *******************************/
  fprintf(out, "dce_server_sec_begin() call\n");
  fflush(out);
  dce_server_sec_begin(dce_server_c_login|dce_server_c_manage_key, &status);
  fprintf(out, "dce_server_sec_begin() return\n");
  fflush(out);
  if (status != error_status_ok) {
    fprintf(out, "Failed in dce_server_sec_begin()\n");
    fflush(out);
    ERROR_CHECK(status, "Cannot sec_begin");
  }

  /********** Only the protocol sequences we want ****************/
  fprintf(out, "dce_server_use_protseq() call\n");
  fflush(out);
  dce_server_use_protseq(NULL, (idl_char *)"ncadg_ip_udp", &status);
  fprintf(out, "dce_server_use_protseq() return\n");
  fflush(out);
  if (status != error_status_ok) {
    fprintf(out, "Failed to specify protocol sequence\n");
    fflush(out);
    ERROR_CHECK(status, "Cannot specify protocol sequence");
  }

  /******* Fill in rest of registration data structures ***********/
  register_data[0].ifhandle = greet_dcedif_v1_0_s_ifspec;
  register_data[0].epv = NULL;  /* use default entry point vector */
  register_data[0].num_types = 0;
  register_data[0].types = NULL;

  /************************** Register the Server *****************/
  fprintf(out, "dce_server_register() call\n");
  fflush(out);
  dce_server_register(dce_server_c_ns_export, /* flag says register server with CDS */
    server_conf,
    register_data,
    &server_handle,
    &status
  );
  fprintf(out, "dce_server_register() return\n");
  fflush(out);
  if (status != error_status_ok) {
    fprintf(out, "Failed dce_server_register. Error %d\n", status);
    fflush(out);
ERROR_CHECK(status, "Can't register server with DCE");
  }

  /******************* Listen for remote procedure calls **********/
  fprintf(out, "Listening...\n");  fflush(out);
  rpc_server_listen(rpc_c_listen_max_calls_default, &status);
  fprintf(out, "Returned from listening...\n");
  fflush(out);
  if (status != rpc_s_ok) {
    fprintf(out, "Failed rpc_server_listen\n");
    fflush(out);
ERROR_CHECK(status, "Can't start listening for calls");
  }

  /*********************** Unregister from DCE *******************/
  fprintf(out, "dce_server_unregister call\n");
  fflush(out);
  dce_server_unregister(&server_handle, &status);
  fprintf(out, "dce_server_unregister return\n");
  fflush(out);
  if (status != error_status_ok) {
    fprintf(out, "Failed dce_server_unregister\n");
    fflush(out);
ERROR_CHECK(status, "Can't unregister server from DCE");
  }

  fprintf(out, "dce_server_sec_done call\n");
  fflush(out);
  dce_server_sec_done(&status);
  fprintf(out, "dce_server_sec_done return\n");
  fflush(out);
  if (status != error_status_ok) {
    fprintf(out, "Failed dce_server_sec_done\n");
    fflush(out);
ERROR_CHECK(status, "Can't do sec_done");
  }

}
```

greet_dced_manager.c

Following are the contents of the **greet_dced_manager.c** file, which contains the implementation of the **greet_dced** interface.

```
/*
 * greet_dced_manager.c
 *
 * Implementation of "greet_dced" interface.
 */

#include <stdio.h>
#include "greet_dced.h"

void
greet_dced(
  handle_t h,
  idl_char *client_greeting,
  idl_char *server_reply
)
{
  printf("The client says: %s\n", client_greeting);

  strcpy(server_reply, "Hi, client!");
}
```

greet_dced_client.c

Following are the contents of the **greet_dced_client.c** file.

```
/*
 * greet_dced_client.c
 *
 * Client of "greet_dced" interface.
 */

#include <stdio.h>
#include <dce/nbase.h>
#include <dce/rpc.h>

#include "greet_dced.h"
```

```
#include "util.h"

int
main(
  int argc,
  char *argv[]
)
{
  rpc_ns_handle_t import_context;
  handle_t binding_h;
  error_status_t status;
  idl_char reply[REPLY_SIZE];

  if (argc < 2) {
    fprintf(stderr, "usage: greet_dced_client <CDS pathname>\n");
    exit(1);
  }

  /*
   * Start importing servers using the name specified
   * on the command line.
   */
  rpc_ns_binding_import_begin(
    rpc_c_ns_syntax_default, (unsigned_char_p_t) argv[1],
      greet_dcedif_v1_0_c_ifspec, NULL, &import_context, &status);
  ERROR_CHECK(status, "Can't begin import");

  /*
   * Import the first server (we could interate here,
   * but we'll just take the first one).
   */
  rpc_ns_binding_import_next(import_context, &binding_h, &status);
  ERROR_CHECK(status, "Can't import");

  /*
   * Make the remote call.
   */
  greet_dced(binding_h, (idl_char *) "hello, server", reply);

  printf("The Greet Server said: %s\n", reply);
}
```

util.c

Following are the contents of the **util.c** file, which contains the error message handling routines for the **greet_dced** server and client.

```
/*
 * util.c
 *
 * Utility routine(s) shared by "greet_dced" client
 * and server programs.
 */

#include <stdio.h>
#include <dce/nbase.h>
#include <dce/dce_error.h>

void
error_exit(
  error_status_t status,
  char *text
)
{
  unsigned char error_text[100];
  int dummy;

  dce_error_inq_text(status, error_text, &dummy);
  fprintf(stderr, "Error: %s - %s\n", text, error_text);
  exit(1);
}
```

util.h

Following are the contents of the **util.h** file, which contains declarations used in the **util.c** file.

```
/*
 * util.h
 *
 * Declarations of utility routine(s) shared by "greet_dced" client
 * and server programs.
 */

#define ERROR_CHECK(status, text) if (status != error_status_ok) \
      error_exit(status, text)

void
error_exit(
  error_status_t status,
  char *text
);
```

greet_dced.install

Following are the contents of the **greet_dced.install** file, which is the **dcecp** install script for the **greet_dced** server.

```
set dir
/users/ your_user_name/src/dce/greet_dced/greet_dced
set cds_dir /.:/subsys/my_company/greet_dced

# Unix and group id of the server process owner
# has to own the keytab file as well
set uid 1265
set gid 1000

# add a PGO for the server
set cmd "user create greet_dced_server -group servers \
    -o osf -password secret -mypwd -dce-"
if {[catch $cmd msg] != 0} {
echo "user create:" $msg
}

# create a directory in CDS and give access to the server
# this will fail if any directory in the chain is not already there
set cmd "directory create $cds_dir"
if {[catch $cmd msg] != 0} {
echo "directory create: " $msg
}

set cmd "acl modify $cds_dir -add {user greet_dced_server rwdit}"
if {[catch $cmd msg] != 0} {
echo "acl modify: " $msg
}

# create a keytab for the server
set cmd "keytab create greet_dced.ktab \
-storage $dir/greet_dced.ktab \
-data {greet_dced_server plain 1 secret}"
if {[catch $cmd msg] != 0} {
echo "keytab create: " $msg
}

# dced create the keytab file with root as its owner
# so we have to chown it, but
# this may require root permission, so it's likely to fail.
set cmd "exec chown $uid $dir/greet_dced.ktab"
if {[catch $cmd msg] != 0} {
echo "chown: " $msg
}

# create the srvrconf object
set cmd "server create greeter_dced \
-program $dir/greet_dced_server \
-entryname $cds_dir/greeter_dced_entry \
-keytabs [attrlist getvalues [keytab show greet_dced.ktab]\
    -type uuid]\
-principals {greet_dced_server} \
-starton explicit \
-directory $dir/exec_dir \
-services { {ifname greet_dced} \
        {interface {3d6ead56-06e3-11ca-8dd1-826901beabcd 1.0}}} \
-uid $uid -gid $gid"
if {[catch $cmd msg] != 0} {
echo "server create: " $msg
}
```

greet_dced.delete

Following are the contents of **greet_dced.delete**, which contains the **dcecp** cleanup script for the **greet_dced** server.

```
set dir /users/ your_user_name/src/dce/greet_dced/greet_dced
set cds_dir /.:/subsys/my_company/greet_dced

catch "server delete greeter_dced"
catch "keytab delete greet_dced.ktab"
catch "directory delete $cds_dir -tree"
catch "user delete greet_dced_server"
```

Makefile

Following are the contents of the **greet_dced** Makefile.

```
######################################################################
#
# Makefile: A generic makefile suitable for building the greet_dced
#      application.
#
#               -77 cols-
#####################################################################

DCEROOT = /opt/dcelocal
CC = /bin/c89
IDL = idl
LIBDIRS = -L${DCEROOT}/usr/lib
LIBS = -ldce -lc_r
LIBALL = ${LIBDIRS} ${LIBS}
INCDIRS = -I. -I${DCEROOT}/share/include
CFLAGS = -g ${INCDIRS} -D_SHARED_LIBRARIES -D__hppa -Dhp9000s800 \
-Dhp9000s700 -D__hp9000s800 -D__hp9000s700 -DHPUX -D__hpux \
-Dunix +DA1.1 -D_HPUX_SOURCE
```

```
IDLFLAGS = -v ${INCDIRS} -cc_cmd "${CC} ${CFLAGS} -c"

all: greet_dced_client greet_dced_server

greet_dced.h greet_dced_cstub.o greet_dced_sstub.o: greet_dced.idl
${IDL} ${IDLFLAGS} greet_dced.idl

greet_dced_client: greet_dced.h greet_dced_client.o util.o \
greet_dced_cstub.o
${CC} -o greet_dced_client greet_dced_client.o \
greet_dced_cstub.o util.o ${LIBALL}

greet_dced_server: greet_dced.h greet_dced_server.o \
greet_dced_manager.o util.o greet_dced_sstub.o
${CC} -o greet_dced_server greet_dced_server.o \
greet_dced_manager.o greet_dced_sstub.o util.o ${LIBALL}

greet_dced_client.c greet_dced_server.c util.c: util.h
greet_dced_manager.c greet_dced_client.c greet_dced_server.c: greet_dced.h

clobber:
rm -f greet_dced.h greet_dced_client greet_dced_client.o \
greet_dced_cstub.o greet_dced_manager.o greet_dced_server \
greet_dced_server.o greet_dced_server_dce.o \
greet_dced_sstub.o server_struct.o greet_dced_server_dce
util.o
```

Chapter 3. DCE Application Messaging

Message generation by distributed programs can be divided into two broad kinds:

- Normal (often user-prompted, client-generated) messages
- Server event messages, containing information about server activity, either normal or extraordinary

Similarly, DCE makes available to applications two messaging APIs:

- The DCE messaging interface
- The DCE serviceability interface

The DCE serviceability interface is designed specifically to route messages of the second type; it is described in “Chapter 4. Using the DCE Serviceability Application Interface” on page 51. Messages in the first category are output using the DCE general-purpose application messaging routines, which are the subjects of this chapter.

Although the two interfaces, broadly speaking, do the same general thing (that is, write messages), their functionality was designed to serve different needs, both of which occur in most distributed applications. Nevertheless, either interface can be used more or less exclusively of the other, if desired.

DCE and Messages

A *message* is a readable character string conveying information about some aspect of a program’s state or activity to a human audience. Messages may be purely informational or they may be intended to be responded to (that is, be interactive). Prompts, error displays, warnings, logs, announcements and program responses are all various kinds of message.

DCE applications can simply use the standard output routines (such as **printf()**, **sprintf()**, and so on) for messaging. However, DCE provides two message interfaces that automatically and transparently take care of many of the special problems that distributed application messaging can give rise to. These interfaces are used by the DCE components themselves to implement their messaging.

Both of the DCE message facilities use XPG4 message catalog files (see the *X/Open Portability Guide*) to hold message text. The message catalogs are generated by a DCE utility (called **sams**) during the application development process, and must be installed in the correct platform-dependent location in order for the DCE messaging library routines to be able to find them (and, consequently, the messages) at runtime.

The main purpose of message catalogs is to allow programs’ message text to be stored and organized (separately from the programs themselves) in a culture- or nationality-specific way. This enables programs to switch their I/O styles and contents to the form appropriate to the geographical location (*locale*) they are running in, simply by using the appropriate catalog. Thus, it is essential to compose catalogs in such a way that each one contains message text appropriate only to a single (same) locale.

Questions such as the proper use of locales, proper language style for messages, where catalogs should be installed, and so on, all fall under the broad topic of *internationalization*, and are not discussed in this chapter. The other important aspect of internationalization, namely character and code set compatibility, is discussed in detail in “Chapter 16. Writing Internationalized RPC Applications” on page 281.

DCE Messaging Interface Usage

Use of the DCE messaging API is very straightforward. In the application code itself, all that is usually required is simply to call one of the output routines, passing it the ID of the message to be output. The messages themselves must first be defined in a text file which must then be processed by the DCE **sams** (symbols and message strings) utility, which generates the message catalog file along with other C source files that contain code necessary to facilitate the additional layer of functionality that DCE has added to the XPG4 message catalog mechanism.

A Simple DCE Messaging Example

The following subsections describe all the steps and code necessary to compile an application that uses the DCE messaging API to print the familiar ″Hello World’’ message.

Defining the Message

For our example, we will define a **sams** file with the minimum contents necessary to print the one brief message we want to display. (Additional information on the use of **sams** can be found in the **sams(1dce)** reference page, which contains comprehensive descriptions of all aspects of the utility.)

Each line in a **sams** file consists of a simple *header* and *value* combination. The *header* indicates the meaning of the value being specified, and *value* is the value itself. A **sams** file for messaging use is normally made up of three parts (although only two parts are needed for the short example in this chapter). The first part consists of a minimum of one line that specifies the name of the *component* (that is, the application) that is to use the messages that will be generated from the file.

Each invocation of **sams** to process a separate **.sams** input file produces a complete set of output files that can be used by the DCE messaging routines to print or log messages as required. These sets of output files are organized by DCE *component*. (In DCE itself, these components are identical to the DCE components: RPC, DTS, and so on; for applications, the categorization of components is determined by the developer.) Each set of output files will have names in which the component name (also determined by the developer) appears.

The component name that you specify at the top of a **sams** file must consist of a three-character (no more, no less) string. For the ″Hello World’’ program we will use the component name **hel**:

```
# Part I of simple sams message file...
component      hel
```

The **hel** string will be used to identify all the files and data structures that **sams** will generate from the file.

The second (and final) part of a **sams** file for DCE messaging consists of a series of records that specify the messages themselves. Each record is delimited by the **start** and **end** keywords. Within each record, a series of keywords identifies the various information that each message consists of or has associated with it.

Our file will contain only one message, the text of which is to be ″Hello World’’. The record that specifies it is as follows:

```
start
code     hello_msg
text     "Hello World"
action    "None required"
explanation  "Greeting message for sample messaging program"
end
```

The keywords specified have the following meanings:

code Identifies the message.

text Specifies the text of the message itself.

explanation
Describes the meaning of the message. The text following this keyword is used to generate the documentation of the component’s messages.

action Describes any action(s) that should be taken in response to the message. The text following this keyword is used to generate the documentation of the component’s messages.

Processing the .sams File

The entire **sams** file for the **hello** program is as follows:

```
# Part I
component   hel

# Part II
start
code     hello_msg
text     "Hello World"
action    "None required"
explanation  "Greeting message for sample messaging program"
end
```

We create the file with these contents and name it **hel.sams**.

A **sams** file containing DCE messaging API message definitions (in other words, a sams file not containing definitions for DCE serviceability API messages) should be compiled by invoking **sams** as follows:

```
sams -o thmc sams_filename
```

where:

-o Introduces output flags as follows:

- **t** Specifies that a file containing source code to generate an in-memory message table be output by **sams**.
- **h** Specifies that a header file defining codes for the message numbers be output by **sams**.
- **m** Specifies that a **.msg** file be output by **sams**.

c Specifies that **sams** call **gencat** (with the **.msg** file as input) to produce a message catalog.

Running the command as shown will result in four files being output:

dcehel.cat
: XPG4 message catalog file created by **gencat**. If you wish to use the message catalog, you must install it yourself.

dcehel.msg
: Message input file for **gencat**.

dcehelmsg.c
: Code defining the in-memory table of message texts. By using this table you can avoid depending on extracting message texts from the message catalog.

dcehelmsg.h
: Header file containing definitions for **dcehelmsg.c**.

The header file should be included in the program source code. The **dcehelmsg.c** module should be compiled and linked with the program object module. The message catalog should be installed in the correct platform-dependent location.

All that remains now is to create a simple C program that calls a DCE messaging routine to output the ″Hello World'' message.

Program Source

The complete source code for **hello.c** is as follows:

```
#include <dce/dce_svc.h>
#include "dcehelmsg.h"

int
main(
  int   argc,
  char  *argv[])
{

  dce_printf(hello_msg);

}
```

To build the application, you simply

- Process the **hel.sams** file with the **sams** command.
- Build and link **hello** from the following modules:
  – **dcehelmsg.c**
  – **hello.c**

When executed, the program will print the following:

```
Hello World
```

This is the text of the **hello_msg** message as defined in the **hel.sams** file.

DCE Messaging and Message Catalogs

The reader may be wondering why, in the previous example, it was not necessary to explicitly open the message catalog before making the call to retrieve and print the message itself. The answer is that **dce_printf()** takes care of this step implicitly. It is able to find the message catalog because the catalog's name is generated from the **component** field in the first part of the **sams** file. Of course, for this to work, the message catalog must be installed in the correct system-defined location before the application is run.

An application may even dispense with the use of installed message catalogs altogether, and use in-memory message tables instead. The necessary code to declare the **sams** file messages as arrays in program memory is contained in the **dce** *cmp***msg.c** file generated with the **sams -o t** option. To initialize the table before using it the application must also call the **dce_msg_define_msg_table()** routine, described in "Message Table Routines" on page 47. The message routines will, when called, attempt to use the application's message catalog; but if it cannot be found, the in-memory table will be used instead.

The DCE Message Interface and sams Input and Output Files

Figure 4 on page 42 shows the relationship of the various files, both source and **sams** output, that go to make up DCE application code that uses the DCE messaging API.

The two parallelogram-shaped objects represent the files that must be created by the developer (you).

Rectangular objects with solid lines represent files that are generated by **sams**; the two ovals represent executable utilities: one is **sams**, the other **gencat** (which is implicitly run by **sams** when message catalogs are generated).

The large rectangular object in dashed lines represents **libdce**, which contains the DCE message API library.

This illustration makes no attempt to show how a DCE application that uses DCE messaging should be compiled and linked, nor how it runs. It is merely a static map of the general place of DCE application messaging in DCE development.

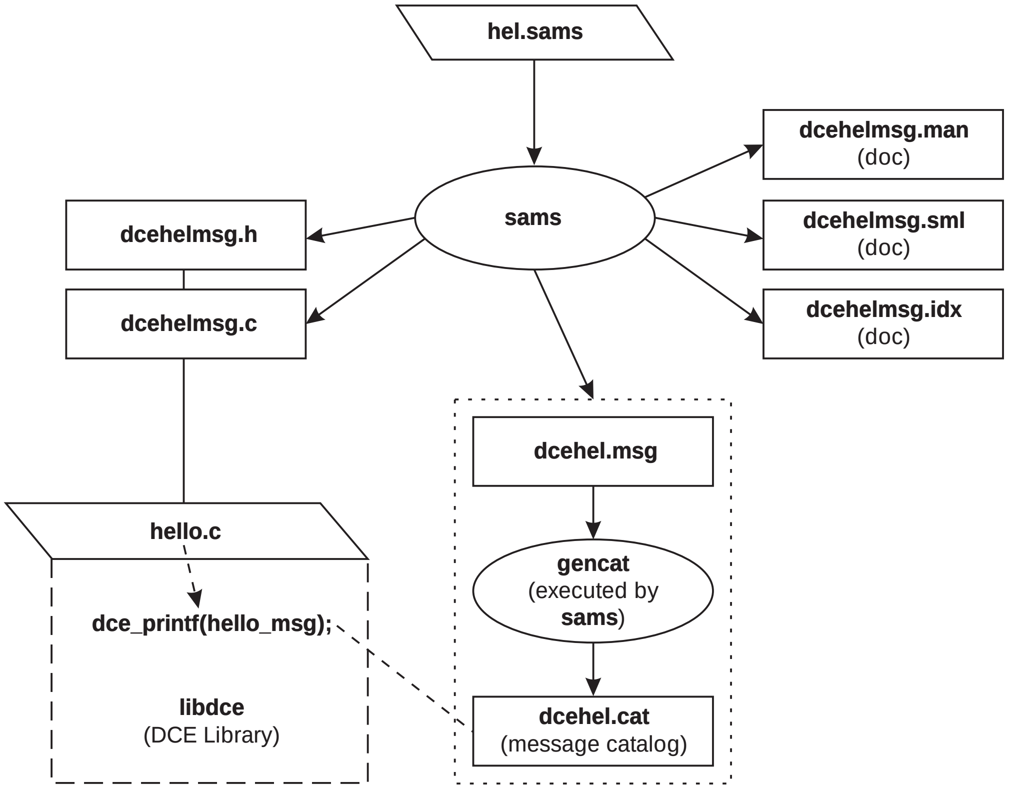


Figure 4. sams and DCE Messages

The **sams** output filenames are made up of the following pieces:

```
tech_name + comp_name +  part_name +  extension
```

where:

tech_name
: Is the technology name (optionally specified at the top of the **hel.sams** file); by default **dce**.

comp_name
: Is the component name (specified at the top of the **hel.sams** file); in this case, **hel**.

part_name
: Is a substring identifying the particular file; for example, **svc** or **msg**, and so on. This piece of the name is omitted from the message catalog filenames (in our example, **dcehel.msg** and **dcehel.cat**).

extension
: Is the file extension (preceded by a **.** (dot) character).

The files **dcehelmsg.man** (generated by **sams -p d hel.sams**) and **dcehelmsg.sml** (generated by **sams -p p hel.sams**), which are shown in Figure 4, were not generated by the following command:

```
sams -o thmc hel.sams
```

They could have been generated by executing this command:

```
sams -o dp hel.sams
```

These are automatically generated documentation files (their nature and purpose were previously described) that have nothing to do with the operation of the interface itself.

A definitive description of **sams** and the contents of **sams** files can be found in the **sams(1dce)** reference page.

DCE Messaging Routines

There are several different DCE messaging routines. It is possible for an application to accomplish all of its messaging tasks with only one or two of these routines (**dce_printf()** and **dce_error_inq_text()**); additional routines allow applications to perform manipulations on message tables, open and close message catalogs explicitly, retrieve messages without printing them, and so on. The complete list of routines is as follows:

- Message output routines

  These routines retrieve and output a specified message. If necessary, the message catalog is opened.

  – **dce_printf()**
  – **dce_fprintf()**
  – **dce_sprintf()**
  – **dce_pgm_printf()**
  – **dce_pgm_sprintf()**
  – **dce_pgm_fprintf()**
- Message retrieval routines

  These routines retrieve a specified message. If necessary, the message catalog is opened.

  – **dce_msg_get_msg()**
  – **dce_msg_get()**
  – **dce_msg_get_default_msg()**
  – **dce_error_inq_text()**
- Message table routines

  Miscellaneous routines for manipulating in-memory message tables.

  – **dce_msg_define_msg_table()**
  – **dce_msg_translate_table(**)
- DCE XPG4 routines

  DCE versions of the XPG messaging routines **catopen()**, **catgets()**, and **catclose()**.

  – **dce_msg_cat_open()**
  – **dce_msg_cat_get_msg()**
  – **dce_msg_get_cat_msg()**
  – **dce_msg_cat_close()**

Generally speaking, routines that retrieve or print messages will first try to get a message from the message catalog file (the routines deduce the correct message catalog from the message ID that is passed to them). Routines will look for the catalog in the current locale’s system-specific location for correctly installed message catalogs.

If the message catalog cannot be found, and an in-memory message table has been defined, the message will be retrieved from there.

The only exception to this message-finding algorithm occurs with **dce_msg_get_default_msg()**, which always attempts to retrieve the in-memory message only.

The following sections describe each of the DCE messaging routines in detail. Complete reference pages for the routines can be found in the *OSF DCE Application Development Reference*.

Message Output Routines

The six message output routines in this group essentially reproduce the functionality of **printf()**, **fprintf()**, and **sprintf()**, with the difference being that they operate on a specified message rather than on a string variable. The routines can be called without any special preparation (but see the descriptions of the three **dce_pgm_** routines).

dce_fprintf()

Retrieves the message text associated with the specified message ID, and prints the message and its arguments on the specified stream. The message is printed *without* a concluding newline; if a newline is desired at the end of the message, then it should be coded (as **\n**) in the message definition in the **sams** file.

The routine determines the correct message catalog and, if necessary, opens it. If the message catalog is inaccessible, and the message exists in an in-memory table, then this message (the default message) is printed. If for any reason the message cannot be retrieved, an error message is printed.

dce_printf()

Performs a **dce_fprintf(**) of the specified message to standard output.

dce_sprintf()

Retrieves the message text associated with the specified message ID, and writes the message and its arguments into an allocated string (which should be freed by the caller). The routine determines the correct message catalog and, if necessary, opens it. If the message catalog is inaccessible, and the message exists in an in-memory table, then this message (the default message) is printed. If for any reason the message cannot be retrieved, an error message is printed.

For example, assume that the following message is defined in an application’s **sams** file:

```
start
code     arg_msg
text     "This message has exactly %d not %d argument(s)"
action    "None required"
explanation  "Test message with format arguments"
end
```

The following code fragment shows how **dce_sprintf()** might be called to write the message (with some argument values) into a string:

```
unsigned char   *my_msg;

my_msg = dce_sprintf(arg_msg, 2, 8);

/* Process my_msg as appropriate...  */

free(my_msg);
```

Of course, **dce_printf()** could also be called to print the message and arguments:

```
dce_printf(arg_msg, 2, 8);
```

dce_pgm_printf()
Equivalent to **dce_printf()**, except that it prefixes the program name to the message (in the standard style of DCE error messages), whereas **dce_printf()** does not. This allows clients (which do not usually use the serviceability interface) to produce error (or other) messages that automatically include the originating application's name. The message is printed with a concluding newline.

Note that the client should call **dce_svc_set_progname()** first to set the desired application name. Otherwise, the default program name will be

```
PID# nnnn
```

where *nnnn* is the process ID of the application making the call.

dce_pgm_sprintf()
Equivalent to **dce_sprintf()**, except that it prefixes the program name to the string (in the standard style of DCE error messages), whereas **dce_sprintf()** does not. Note that the client must call **dce_svc_set_progname()** first to set the desired application name.

Otherwise, the default name is

```
PID# nnnn
```

where *nnnn* is the process ID of the application making the call.

dce_pgm_fprintf()
Equivalent to **dce_fprintf()**, except that it prefixes the program name to the string (in the standard style of DCE error messages), whereas **dce_fprintf()** does not. The message is printed with a concluding newline.

Note that the client must call **dce_svc_set_progname()** first to set the desired application name. Otherwise, the default name is

```
PID# nnnn
```

where *nnnn* is the process ID of the application making the call.

dce_error_inq_text()
Opens a message catalog, extracts a message identified by a message ID, and places the message in the space pointed to by the *text* parameter. If the message catalog is inaccessible, and there is a default message in memory, the default message is copied into the space passed. If neither the catalog nor the default message is available, a status code is placed in the status output parameter and the message is returned as a hexadecimal number; the routine always returns a printable message.

This routine existed in prior releases of DCE and has been modified for DCE Version 1.1 to use the default message arrays. Programs prior to Version 1.1 that use the routine do not need to be modified.

For example, assume that the following message is defined in an application's **sams** file:

```
start
code     error_msg
text     "Error: %s"
action    ""
explanation  "DCE error status message"
end
```

The following code fragment shows how **dce_error_inq_text()** could be used to retrieve the error status received from a DCE routine:

```
dce_error_string_t error_string;
unsigned32     status;
int         error_inq_status;
uuid_t       type_uuid, obj_uuid;


<. . .>

rpc_object_set_type(&obj_uuid, &type_uuid, &status);
if (status != rpc_s_ok)
{
 dce_error_inq_text(status, error_string, \
           &error_inq_status);
 dce_printf(error_msg, error_string);
}
```

Message Retrieval Routines

The following three routines retrieve messages, but do not print them.

dce_msg_get_msg()
Retrieves a message (identified by a global message ID) from a message catalog, and returns a pointer to a **malloc()**'d space containing the message. The routine determines the correct message catalog and opens it. If the message catalog is inaccessible, and the message exists in an in-memory table, then this message (the default message) is returned in the allocated space. If neither the catalog nor the default message is available, an error status code is placed in the status output parameter.

The following code fragment shows how **dce_msg_get_msg()** might be called to retrieve the ″Hello World'' message defined in the example program earlier in this chapter:

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include "dcehelmsg.h"

unsigned char    *my_msg;
unsigned32     status;


<. . .>

my_msg = dce_msg_get_msg(hello_msg, &status);
printf("Message is: %s\n", my_msg);
free(my_msg);
```

dce_msg_get()

This is a convenience form of **dce_msg_get_msg()**. If it fails, it does not pass back or return a status code, but instead fails with an assertion error, that is, aborts the calling process.

The following code fragment shows how the routine might be called to retrieve the ″Hello World'' message defined in the example program earlier in this chapter:

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include "dcehelmsg.h"

unsigned char    *my_msg;


<. . .>

my_msg = dce_msg_get(hello_msg);
printf("Message is: %s\n", my_msg);
free(my_msg);
```

dce_msg_get_default_msg()

Retrieves a message (identified by a global message ID) from an in-memory message table and returns a pointer to static space containing the message retrieved. If the default message is not available, an error status code is placed in the status output parameter.

The following code fragment shows how **dce_msg_get_default_msg()** might be called to retrieve the in-memory copy of the ″Hello World'' message defined in the example program earlier in this chapter:

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include "dcehelmsg.h"

unsigned char    *my_msg;
unsigned32     status;


<. . .>

my_msg = dce_msg_get_default_msg(hello_msg, &status);
printf("Message is: %s\n", my_msg);
```

Note that, in order for this call to be successful, **dce_msg_define_msg_table()** must first have been called to set the table up in memory. For an example of how this is done, see the following section.

Message Table Routines

The two routines in this group are intended to be used to perform manipulations on the in-memory message table.

The in-memory table is implemented with code generated by **sams** and contained in the **dce** *cmp***msg.c** module (where *cmp* is the component name of the application, as specified in the **component** field in part I of the **sams** file). This file must then be compiled and linked with the application, and **dce_msg_define_msg_table()** is called at runtime to set up the table.

Note that, even if an in-memory table is defined, the DCE messaging routines still will always attempt first to extract the specified message from the message catalog, and only if unsuccessful will they revert to the in-memory table. The exception to this rule is **dce_msg_get_default_msg()**, which always attempts to retrieve the in-memory message only.

dce_msg_define_msg_table()

Installs a default in-memory message table accessible to DCE messaging routines. This routine is intended to be used by programs that load all messages from a catalog into memory in order to avoid file access overhead on message retrieval.

The following code fragment shows how **dce_msg_define_msg_table()** might be called to set up an in-memory message table consisting of the contents of the messages defined in **hel.sams** earlier in this chapter:

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include "dcehelmsg.h"

unsigned32     status;


<. . .>

dce_msg_define_msg_table(hel_msg_table,
 sizeof(hel_msg_table) / sizeof(hel_msg_table[0]),
             &status);
```

dce_msg_translate_table()

Makes a new copy of the specified in-memory message table (that is, updates the table with the contents of a message table, which has changed because of a change in locale).

Note that this routine will fail if the message catalog is not installed or if **LANG** is not properly set, since the update depends on accessing the contents of the message catalog (unlike the initial table setup, which is done from the code in the **dce** *cmp***msg.c** file).

The following code fragment shows how **dce_msg_translate_table()** might be called to translate the in-memory table that was set up by the call to **dce_msg_define_msg_table()** in the previous example:

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include <locale.h>
#include "dcehelmsg.h"

char        *loc_return;
unsigned32     status;


<. . .>

loc_return = setlocale(LC_MESSAGES, "C");
dce_msg_translate_table(hel_msg_table,
 sizeof(hel_msg_table) / sizeof(hel_msg_table[0]),
            &status);
```

DCE XPG4 Routines

The four routines in this group provide DCE versions of functionality of the XPG messaging routines **catopen()**, **catgets()**, and **catclose()**.

dce_msg_cat_open()

(DCE abstraction over **catopen()**) Opens a message catalog identified by a message ID. The routine returns a handle to the open catalog from which messages will be extracted. This routine is intended for use by applications (such as user interface programs) that display many messages from a particular catalog.

The routine will fail if the message catalog is not installed or if **LANG** is not properly set.

The following code fragment shows how **dce_msg_cat_open()** might be called to open the message catalog containing the ″Hello World'' message defined for the example application earlier in this chapter:

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include "dcehelmsg.h"

dce_msg_cat_handle_t  hel_msg_handle;
unsigned32       status;


<. . .>

hel_msg_handle = dce_msg_cat_open(hello_msg, &status);
```

dce_msg_cat_get_msg()

(DCE abstraction over **catgets()**) Retrieves a message from an open catalog. If the message is not available, returns NULL.

The routine will fail if the message catalog is not installed or if **LANG** is not properly set.

The following code fragment shows how **dce_msg_cat_get_msg()** might be called to retrieve the ″Hello World'' message. Note that the message catalog must first be opened.

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include "dcehelmsg.h"

dce_msg_cat_handle_t  hel_msg_handle;
unsigned32       status;
unsigned_char_t     *msg;


<. . .>

hel_msg_handle = dce_msg_cat_open(hello_msg, &status);
msg = (unsigned_char_t *)dce_msg_cat_get_msg(hel_msg_handle,
                      hello_msg,
                      &status);
printf("Message from dce_msg_cat_get_msg == %s\n", msg);
```

dce_msg_get_cat_msg()

Convenience form of previous routine. Opens a message catalog, extracts a message identified by a global message ID, and returns a pointer to **malloc()**'d space containing the message. If the message catalog is inaccessible, returns an error.

The routine will fail if the message catalog is not installed or if **LANG** is not properly set.

The following code fragment shows how **dce_msg_get_cat_msg()** might be called to retrieve the ″Hello World'' message:

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include "dcehelmsg.h"

unsigned32       status;
unsigned_char_t     *msg;


<. . .>
```

```
msg = dce_msg_get_cat_msg(hello_msg, &status);
printf("Message from dce_msg_get_cat_msg == %s\n", msg);
```

dce_msg_cat_close()

(DCE abstraction over **catclose()**) Closes the catalog specified by *handle*.

The following code fragment shows how **dce_msg_cat_close()** might be called to close the message catalog containing the ″Hello World'' message:

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include "dcehelmsg.h"

dce_msg_cat_handle_t  hel_msg_handle;
unsigned32       status;


<. . .>

dce_msg_cat_close(hel_msg_handle, &status);
```

Chapter 4. Using the DCE Serviceability Application Interface

DCE serviceability was originally developed simply as a way of standardizing server messages. The goal of its design was to make sure that all situations requiring human intervention that can be encountered by a server are documented and identified (both by the server when reporting them, and by the documentation when explaining them) in a standard coordinated way so that system administrators can easily determine the proper corrective action to take in response. Both the server message text and the relevant documentation are derived from the same source (that is, a **.sams** input file), which minimizes the possibility of any discrepancies appearing between the two.

The serviceability component is used by the DCE components (RPC, DTS, Security, and so on) for their server messaging, and it is made available as an API for use by DCE application programmers who wish to standardize their applications' server messaging. (The DCE components are required to use the serviceability routines, but applications are not.)

Overview

Serviceability uses XPG4 message catalogs to hold message text, but it adds an additional layer to the XPG4 functionality. The message catalogs and other required data (and documentation) files are generated by a utility called **sams** (symbols and message strings). Its input is a text file that establishes some organizational information about the program that is to use the messages, followed by a series of specifications of the messages themselves.

Each message specification contains, along with the message text itself, a detailed explanation of the situation in which the message will be displayed, together with a description of the action required, where applicable, to correct the situation. Part of the output of **sams** thus consists of automatic documentation of all the messages writable via the serviceability API. This output was used as the basis of the contents of the *OSF DCE Problem Determination Guide* for the DCE component server messages.

Messages also have one or more *attributes* specified in the **sams** input file. The attributes fall into three broad categories: those that indicate message *severity*, those that specify message *routing*, and those that specify some *action* (usually some form of program exit) that should be taken immediately after the message is written. The effect of the presence of a severity attribute is to cause the message text to contain a severity-identifying string when displayed or written. The effect of the presence of a routing attribute is to cause the message to be routed by default to one of a couple of standard destinations (more flexible routing is available dynamically). The effect of the presence of an action attribute is to cause the program to terminate execution in one of three ways as soon as the message has been written or displayed, or to cause a special short form of the message to be generated.

The serviceability API can also be used by DCE applications. The advantage in using it consists mainly in the following:

- It allows all application messaging to be routed uniformly, on the basis of the severity of the message and the functional part of the program originating the message.
- It allows application messages to be made self-documenting.

Serviceability also contains facilities for debug messaging, which can be compiled in or out of executables and which can be activated and routed by component at nine different levels.

How Programs Use Serviceability

The DCE serviceability mechanism uses XPG4 message catalogs to hold message text. Additional files contain the messages' associated documentation and other extra information used by the mechanism. All of these files, including the message catalog, are generated in a single step by running the DCE **sams** utility. The input to **sams** is a single **sams** file that is written by the developer, and which contains all the necessary information (text, documentation, additional information) for each message. The message catalogs and associated information generated by **sams** are then accessed whenever **dce_svc_printf()** or one of the other serviceability routines is called to print or log a message.

Thus, the result of converting a program to use serviceability will essentially be that all **printf()**, **fprintf()**, and other such routines will be replaced by calls to **dce_svc_printf()** or one of the related serviceability routines. For example, a line of code such as the first one that follows would be replaced by the second:

```
fprintf(stderr, "File %s not found\n", filename);

dce_svc_printf(DCE_SVC(cmp_svc_handle, ""), cmp_s_server, \
 svc_c_sev_error, cmp_s_file_not_found, filename);
```

where the constants *cmp*_**s_server** and *cmp*_**s_file_not_found** were generated by **sams**, and identify the server subcomponent of the application and the message to be written, respectively. The *cmp*_**svc_handle** constant is the application's handle to its serviceability message tables and other necessary data; *cmp*_**s_server** is actually an index to a subtable within this dynamically generated area, and *cmp*_**s_file_not_found** is the index of the message text within the subtable.

By convention, *cmp* is a three-character code identifying the application as a whole; serviceability uses it to group all of an application's message and table data together. Specifying **svc_c_sev_error** gives the message the severity of error; the significance of severity in serviceability will be explained in the following sections. **DCE_SVC()** is a macro that helps simplify the coding of **dce_svc_printf()** calls; as will be seen, another macro mechanism can be used to make the calls much simpler still.

Simple Serviceability Interface Tutorial

In this section, we'll see how to go about creating a simple C program that uses the serviceability interface to print the familiar ″Hello World'' message.

All that is necessary to do this is to replace the first call that follows with something like the second:

```
printf("Hello World\n");

dce_svc_printf(hello_world_message);
```

However, making the **dce_svc_printf()** call requires the following preliminary steps:

1. Defining the message in a **sams** file.

2. Processing the **sams** file to obtain a set of files that contain code used by the serviceability routines.
3. Coding some serviceability initialization calls into the C program itself.
4. Coding the **dce_svc_printf()** call.

The next several sections describe each of these steps.

Defining the Message

In order to print any message through the serviceability interface, we must first define the message in a **sams** file and process the file with the **sams** utility. For the **hello_svc** program, we will define a **sams** file with the bare minimum contents necessary. Additional information on the use of **sams** can be found in the **sams(1dce)** reference page.

Each line in a **sams** file consists of a simple *header* and *value* combination. The *header* indicates the meaning of the value being specified, and *value* is the value itself. A **sams** file for serviceability use is made up of three parts. The first part consists of a minimum of one line that specifies the name of the *component* (that is, the application) that is to use the messages that will be generated from the file.

The component name that you specify at the top of a **sams** file must consist of a three-character (no more, no less) string. For the ″Hello World'' program, we will use the component name **hel**:

```
# Part I of simple sams file ...
component      hel
```

The **hel** string will be used to identify all the files and data structures that **sams** will generate from the file.

The second part of the **sams** file contains some additional serviceability-specific information about the message data structures that will be generated. (This information is necessary if the **sams** file is intended for serviceability use because **sams** is also used to generate message files for general, nonserviceability use.)

This part of the file specifies the names of the serviceability *table* and the serviceability *handle*. It also contains a list of the component's *subcomponents*. A subcomponent consists of a distinct functional module of executing code. For example, most distributed applications would have a basic server subcomponent, a reference monitor subcomponent that would handle authorization decisions, and one or more subcomponents that would contain the application's particular functionality.

The serviceability interface finds a component's messages in one or more subtables, each one associated with a subcomponent. When the message is displayed or written, the associated subcomponent name is written in a field of the message. This allows messages to be distinguished during routing or other processing on the basis of the subcomponent with which they are associated.

Following is what the second part of our simplified sample **sams** file looks like. We call the serviceability table **hel_svc_table**, and we call the serviceability handle **hel_svc_handle**. Although we have used the three-letter component code **hel** in these names, we were under no obligation to do so; we could have named the table and the handle anything we wanted. (We will need to know both of these names when we make the call in the application to initialize the interface in preparation for displaying messages.)

A component must have at least one subcomponent specified in its **sams** file. Subcomponents are specified in this part simply by supplying their *table index*, their *name*, and their *descriptive id* in a series of separate lines, one per subcomponent and each one beginning with the **sub-component** keyword, between a set of **start** and **end** keywords:

```
# Part II
serviceability table hel_svc_table handle hel_svc_handle
start
subcomponent  hel_s_main   "main" hel_i_svc_main
end
```

In our example,

hel_s_main
is the table index name for the subcomponent. Serviceability routines need this name in order to locate and print any of the subcomponent's messages.

main is the name of the subcomponent, specified in quotes.

hel_i_svc_main
is a name that will be used (later on in the file) to identify a message that describes the subcomponent.

(Note that **sams** assigns values to all of these indexes automatically.)

The third (and final) part of the **sams** file consists of a series of records that specify the messages themselves. Each record is delimited by the **start** and **end** keywords. Within each record, a series of keywords identifies the various information that each message consists of or has associated with it.

Our file will contain only one message, the text of which is to be ″Hello World''. The record that specifies it is as follows:

```
# Part III
start
code        hel_s_hello
subcomponent    hel_s_main
attributes     "svc_c_sev_notice | svc_c_route_stderr"
text        "Hello World"
explanation     "A short informational greeting"
action       "None required."
end
```

The keywords specified have the following meanings:

start Marks the beginning of a message definition. This keyword can optionally be followed by various values.

- A *number* following the keyword specifies that the ID that is generated by **sams** for the message should be based on (*number* multiplied by 100). This allows the ID numbers of messages that belong to the same subcomponent of an application to be in the same numerical subseries (**collection**), even if additional messages for subcomponents have to be added later on. If each subcomponent's first message is **start**ed with a **collection** number that allows for enough extra ID space in the previous subcomponent to accommodate a reasonable number of future additional definitions, then each subcomponent's ID series will be able to maintain its unbroken series.

As mentioned, the default size of a **collection** number is 100. Thus, the following **collection** specification is interpreted as ″200'':

```
start       2
```

To change the default **collection** size, specify

```
collection size dddd
```

(where *dddd* is the **collection** size you desire) in a separate line in Part 1 of the **sams** file.

code Identifies the message.

sub-component
Identifies the subcomponent that will use the message. (This must also have been defined in Part II of the **sams** file.)

attributes
Specifies various things about the message: what kind of message it is, how it is to be routed, and so on. Multiple attributes are specified by ORing their values together. In the example shown, the message has the *severity* attribute **svc_c_sev_notice**, and the *routing* attribute **svc_c_route_stderr**; the latter forces the message to be routed to **stderr** whenever it is written by a serviceability routine.

text Specifies the text of the message itself.

explanation
Describes the meaning of the message. The text following this keyword is used to generate the documentation of the component's messages.

action Describes any action(s) that should be taken in response to the message. The text following this keyword is used to generate the documentation of the component's messages.

Not all the possible keywords are illustrated in our example, and, of those illustrated, only **code** and **text** are required in all circumstances. In the example, **explanation** and **action** have been specified because it is simpler at this point to do so than to leave them out, and **attributes** and **sub-component** have been specified for reasons that will be made clear later on.

This final part of the **sams** file also contains a series of one or more records that specify messages identifying each of the subcomponents themselves. Since our application has only one subcomponent, it contains only one such subcomponent-identifying message:

```
# Part IIIa
# Messages for serviceability table

start        !intable undocumented
code        hel_i_svc_main
text        "hello_svc main"
end
```

The keywords have the same meanings as they did in the ″Hello World'' message. A couple of flags have been specified after the **start** keyword. The first will cause the message to *not* be generated in the message table, and the second means that the message does not need any **explanation** or **action** text associated with it. By specifying **undocumented** (with **intable**, to cause the message to actually be generated even though it was to be undocumented) for the “Hello” message, we could have eliminated the **explanation** and **action** keywords there also.

Processing the sams File

The entire **sams** file for the **hello_svc** program is as follows:

```
# Part I
component      hel
table        hel_msg_table

# Part II
serviceability table hel_svc_table handle hel_svc_handle
start
subcomponent  hel_s_main   "main" hel_i_svc_main
end
# Part III
start
code        hel_s_hello
subcomponent    hel_s_main
attributes     "svc_c_sev_notice | svc_c_route_stderr"
text        "Hello World"
explanation     "?"
action       "None required."
end
# Part IIIa
start        !intable undocumented
code        hel_i_svc_main
text        "hello_svc main"
end
```

We create the file with these contents and name it **hel.sams**. It can be processed with the simple command that follows:

```
sams hel.sams
```

Running the command as shown will result in ten files being created:

dcehel.cat
XPG4 message catalog file created by **gencat**. If you wish to use the message catalog, you must install it yourself. Its proper location is platform dependent.

dcehel.msg
Message input file for **gencat**.

dcehelmac.h
Defines *convenience macros* for use with the serviceability interface to write serviceability messages.

dcehelmsg.c
Code defining the in-memory table of message texts. By using this table, you can avoid depending on extracting message texts from the message catalog.

dcehelmsg.h
Header file containing definitions for **dcehelmsg.c**.

dcehelmsg.sml
Code for a *OSF DCE Problem Determination Guide* subsection documenting the messages.

dcehelmsg.man
Code for a reference page subsection documenting the messages.

dcehelmsg.idx
Code for building an index for the *OSF DCE Problem Determination Guide* subsection.

dcehelsvc.c
Code defining the serviceability table. (This is a separate table containing the serviceability subcomponent identifying messages specified at the end of the **sams** file.)

dcehelsvc.h
Header file containing definitions for **dcehelsvc.c**.

Of these files, only the following are needed for the **hello_svc** program:

dcehelmac.h
Contains convenience macro code.

dcehelmsg.c
Contains in-memory message table code.

dcehelmsg.h
Contains definitions for in-memory message table code.

dcehelsvc.c
Contains serviceability message table code.

dcehelsvc.h
Contains definitions for serviceability message table code.

The three header files should be included into the program source code. The **dcehelmsg.c** and **dcehelsvc.c** modules should be compiled and linked with the program object module.

All that remains now is to create a simple C program that calls the necessary serviceability routines to output the ″Hello World'' message.

Coding the Serviceability Calls

The bare minimum required to initialize the serviceability interface and use it to display our message is

- Call **dce_svc_register()** to get a serviceability handle that we can pass to serviceability message routines.
- Call **dce_msg_define_msg_table()** to set up the in-memory message table.
- Call **dce_svc_printf()** to print the message.

To call **dce_svc_register()**, you must declare the serviceability handle that you defined in **hel.sams**:

```
#include "dcehelsvc.h"

          <. . .>

dce_svc_handle_t hel_svc_handle;
unsigned32    status;

          <. . .>

hel_svc_handle = dce_svc_register(hel_svc_table, \
 (idl_char*)"hel", &status);
if (status != svc_s_ok)
{
  printf("dce_svc_register failed\n");
  exit(1);
}
```

This call is the only initialization we need if we have installed our message catalog and are willing to depend on the message(s) being extracted from there. However, if we wish to have the messages available in program memory (and thus not depend on the catalog's being correctly installed), then we have to call **dce_msg_define_msg_table()** to initialize the in-memory table, as follows:

```
#include <dce/dce_msg.h>
#include "dcehelmsg.h"

       <. . .>

dce_msg_define_msg_table(hel_msg_table,
   sizeof(hel_msg_table) / sizeof(hel_msg_table[0]),
             &status);
if (status != svc_s_ok)
{
  printf("dce_svc_define_msg_table failed\n");
  exit(1);
}
```

Now we can call **dce_svc_printf()** to print the message, as follows:

```
#include "dcehelmac.h"

       <. . .>

dce_svc_printf(HEL_S_HELLO_MSG);
```

Note the argument **HEL_S_HELLO_MSG**, which we did not define in the **hel.sams** file. **HEL_S_HELLO_MSG** is, in fact, a macro that was generated by **sams** from our definition for the **hel_s_hello** message, as you can see from the following code:

```
start
code        hel_s_hello
subcomponent    hel_s_main
attributes     "svc_c_sev_notice | svc_c_route_stderr"
text        "Hello World"
explanation     "?"
action       "None required."
end
```

The macro automatically generates the long argument list that must be passed to **dce_svc_printf()** to get it to print the message. The code for this convenience macro is contained in **dcehelmac.h**.

A convenience macro is generated for every message in a **sams** file that has both **sub-component** and **attributes** specified. The macro's name is formed from the uppercase version of its **code** value (as specified in the **sams** file), with the string **_MSG** appended.

The complete source code for **hello_svc.c** is as follows:

```
#include <dce/dce.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
```

```
#include <dce/utctypes.h>
#include <pthread.h>
#include <dce/dce_msg.h>

#include "hel_svc.h"
#include <dce/dcesvcmsg.h>
#include "dcehelmsg.h"
#include "dcehelsvc.h"
#include "dcehelmac.h"

int main(int argc,
     char *argv[])
{

  dce_svc_handle_t  hel_svc_handle;
  unsigned32     status;

  hel_svc_handle = dce_svc_register(hel_svc_table, \
   (idl_char*)"hel", &status);
  if (status != svc_s_ok)
  {
      printf("dce_svc_register failed\n");
      exit(1);
  }

  dce_msg_define_msg_table(hel_msg_table,
     sizeof(hel_msg_table) / sizeof(hel_msg_table[0]),
               &status);
  if (status != svc_s_ok)
      printf("dce_svc_define_msg_table failed \
            -- will use catalogs\n");

  dce_svc_printf(HEL_S_HELLO_MSG);

}
```

Building and Running the Program

To build the application, you simply perform these steps:

1. Process the **hel.sams** file with the **sams** command
2. Build and link **hello_svc** from the following modules:
   - **dcehelmsg.c**
   - **dcehelsvc.c**
   - **hello_svc.c**

Fields of a Serviceability Message

When executed, the program prints a message similar to the following:

```
1994-04-05-20:13:34.500+00:00I----- PID#9467 \
     NOTICE hel main hello_svc.c 47 0xa444e208
Hello World
```

This message is made up of the following fields:

time inaccuracy process_ID severity component subcomponent src_file src_line thread_ID text

Where the field names have the following meanings:

time The time that the message was written, in ISO format:

CCYY-MM-DD-hh:mm:ss.fff[+|-]II:ii

Where the digit groups represent:

CCYY	Century and year
MM	Month
DD	Day
hh	Hour
mm	Minutes
ss	Seconds
fff	Fractions of second
II:ii	Time inaccuracy expressed in hours and minutes

The final groups represent a time differential factor (expressed in hours and minutes), followed by an inaccuracy component. For further information on time format, see “Chapter 20. Introduction to the Distributed Time Service API” on page 459.

process_ID
: The process ID of the program that wrote the message (**PID#9467** in the example). If **dce_svc_set_progname()** had been called to establish the application’s program name, that name would appear in this field instead of the process ID. See “Basic Server Calls” on page 64 for further information.

severity
: The severity level of the message (**NOTICE** in the example).

component
: The component name of the program that wrote the message (**hel** in the example).

subcomponent
: The subcomponent that wrote the message (**main** in the example; note that this program has only one subcomponent).

src_file
: The name of the C source file in which the **dce_svc_printf()** call was executed.

src_line
: The line number, in the source file, at which the **dce_svc_printf()** call is located.

thread_ID
: The thread ID of the thread that wrote the message, expressed as a hexadecimal number (**0xa444e208** in the example).

text
: The text of the message (**Hello world** in the example).

Serviceability Input and Output Files

Figure 5 on page 61 shows the relationship of the various files, both source and **sams** output, that go to make up the **hello_svc** application.

The two parallelogram-shaped objects represent the files that must be created by the developer (you).

Rectangular objects with solid lines represent files that are generated by **sams**; the two ovals represent programs: one is **sams**, the other **gencat** (which is implicitly run by **sams** when message catalogs are generated).

The large rectangular object in dashed lines represents **libdce**, which contains the serviceability library.

The diagram makes no attempt to show how **hello_svc.c** itself is compiled and linked, nor how it runs. It is just a static map of the general place of serviceability in DCE development.

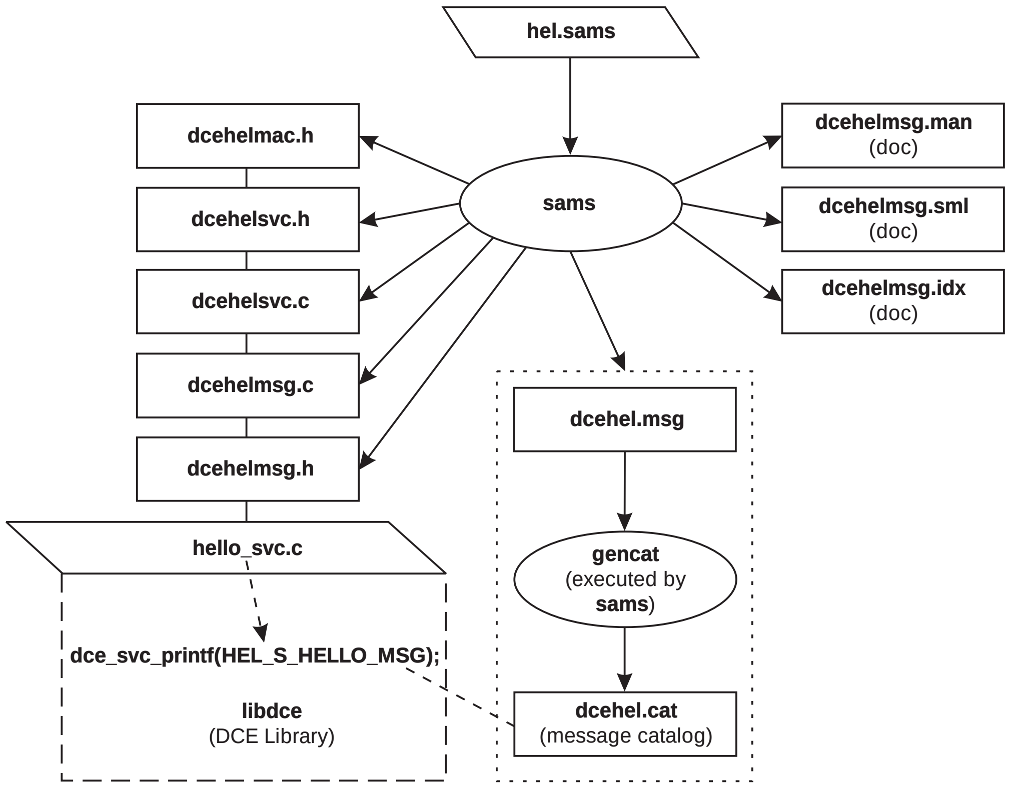


Figure 5. Serviceability and DCE Applications

The **sams** output filenames are constructed as follows:

```
tech_name.comp_name.part_name.extension
```

where:

tech_name
: Is the technology name (optionally specified at the top of the **hel.sams** file); by default it is **dce**.

comp_name
: Is the component name (specified at the top of the **hel.sams** file); in this case, **hel**.

part_name
: Is a substring identifying the particular file; for example, **svc** or **msg**, and so on. This piece of the name is omitted from the message catalog filenames (in our example, **dcehel.msg** and **dcehel.cat**).

extension
Is the file extension (preceded by a **.** (dot) character).

Because we executed the simplest form of the **sams** command (that is, without specifying any output flags), the full repertory of **sams** output files was created, even though the following files were not needed for our application:

- **dcehel.msg** and **dcehel.cat**

  The file **dcehel.msg** is input to **gencat** when it is invoked by **sams** to create **dcehel.cat**, the message catalog. (Although our example application used in-memory tables, the serviceability routines always attempt to use the message catalog first.)
- **dcehelmsg.man** and **dcehelmsg.sgm**

  These are automatically generated documentation files (their nature and purpose were previously described) that have nothing to do with the operation of the interface itself.

The many additional features of serviceability will be described in the following sections. A definitive description of **sams** and the contents of **sams** files can be found on the **sams(1dce)** reference page.

Integrating Serviceability into a Server

The purpose of the preceding tutorial was simply to give a brief taste of what it feels like to use the interface. The main task involved in using serviceability does not, however, lie in mastering the mechanics of the interface, but rather in understanding the purpose of handling server messages in this way, and then applying this understanding in order to develop an effective and serviceable messaging strategy for one's own application.

Serviceability Strategy

The serviceability mechanism is designed to be used mainly for server informational and error messaging—that is, for messages that are of interest to those who are concerned with server maintenance and administration (in the broadest sense of these terms). The essential idea of the mechanism is that all server events that are significant for maintaining or restoring normal operation should be reported in messages that are made to be self-documenting so that (provided all significant events have been correctly identified and reported) users and administrators will by definition always be able to learn what action they should take whenever anything out of the ordinary occurs. User-prompted, interactive, client-generated messaging should be handled through the DCE messaging interface, which is described in "Chapter 3. DCE Application Messaging" on page 37.

It follows that serviceability is not just an interface; it is partly a state of mind. The first thing that a developer who wishes to use serviceability should do is examine his or her server code with a view to identifying all the *event points* that should be covered by serviceability calls. Once these have been determined, the **sams** file (containing the message definitions) should be written; the last step will be to insert the messaging calls into the code.

Components and Subcomponents

The very first step in incorporating serviceability into a server is to analyze it into functionally discrete modules (called *subcomponents*), each of which will usually be associated with a separate set of messages.

The program itself is regarded as the component. The main significance of subcomponents is that each one uses a separate part of the message table generated by **sams**, and every message is identified both by component and by subcomponent; message routing and the level of debug messaging can be specified separately by subcomponent.

Identifying Event Points

Once you have established the subcomponent organization of the server application, you can begin the work of identifying all the points in the server code at which events occur or can occur that require serviceability messaging.

Following is a list of the events and kinds of events that should be reported through the serviceability interface:

- Server startup

  Servers should report when they are started, when they have completed their initialization, and when they are ready to perform work. They should also indicate when they are going offline.
- Server exit

  All fatal exits should be reported as fatal errors, using the **svc_c_sev_fatal** severity attribute in a call to **dce_svc_printf()**. In other words, **exit()** or **abort()** should not be called directly; this ensures that all such fatal errors will be logged. For an explanation of severity level attributes, see “Basic Server Calls” on page 64.
- Other fatal errors

  Errors that make it impossible to proceed should be detected as close as possible to the point where the actual failure occurred. This class of error includes such impossible conditions as failure to successfully allocate memory, failure to open a configuration file for reading, failure to open a log file for writing, and so on.
- Impaired efficiency

  Conditions that may indicate system-level malfunction or poor performance should be reported as warnings. An example of such a situation (from one of the DCE components) would be the RPC runtime detecting that it is having to make an excessive number of retransmits.
- Significant routine activity

  Routine administrative actions should be reported as informational (**svc_c_sev_notice**) messages. Such activity includes creation, modification and deletion of tickets, threads, files, sockets, RPC endpoints, or other objects; message transfer, including name lookup, binding, and forwarding; directory maintenance (including caching, advertising, skulking, and replication); and database maintenance (including replication or synchronization).
- Data input syntax errors

  Routines that process data that could have been entered from a keyboard should fail gracefully (and not core dump, for example) if the data is syntactically incorrect. Serviceability can be used to report this kind of failure.

Once you have identified the points in your code that should be reported with serviceability messaging, the next step is to define the messages themselves (in the **sams** file) and code the serviceability calls. The serviceability features of **sams** files were described previously; the following sections describe the various parts of the serviceability interface itself.

Using the serviceability interface to report errors ensures that the error codes used will be unique within DCE.

Application Use of Serviceability

The following subsections describe in detail the various elements of the serviceability API and what you can do with them.

Complete reference pages for all the serviceability routines can be found in the *OSF DCE Application Development Reference*.

Basic Server Calls

The basic serviceability routines are the following:

- DCE_SVC_DEFINE_HANDLE()

  This is a macro that can be used instead of **dce_svc_register()** to register a table (it does this by means of a global variable created at compile time). It could have been used in the **hello_svc.c** code as follows, with exactly the same results as from using **dce_svc_register()**:

  ```
  DCE_SVC_DEFINE_HANDLE(hel_svc_handle, hel_svc_table,
  "hel");
  /*            handle      |     |  */
  /*                  table     |  */
  /*                    component name
  */
  ```

  Note that either **DCE_SVC_DEFINE_HANDLE()** or **dce_svc_register()** *must* be called before the interface can be used.

- **dce_svc_register()**

  This is the function call for registering a serviceability message table. Either it or **DCE_SVC_DEFINE_HANDLE()** must be called before any routines can be called to display or log messages. An example of its use can be seen in the illustrated **hello_svc.c** code.

- **dce_svc_unregister()**

  This is the function call for destroying a serviceability handle. Calling it closes any open message routes and frees all allocated resources associated with the handle. However, it is not usually necessary to call this routine since the normal process exit will perform the required cleanup.

  The routine could have been called at the end of the **hello_svc.c** application as follows:

  ```
  dce_svc_unregister(hel_svc_handle, &status);
  ```

  where **hel_svc_handle** is the serviceability handle that was originally returned by the call to **dce_svc_register()**, or filled in by the **DCE_SVC_DEFINE_HANDLE()** call.

- **dce_svc_set_progname()**

This function sets the application's *program name*, which is included in all messages. In this way, multiple programs can write messages to the same file and the messages will remain distinguishable.

For example, this routine could have been called in the **hello_svc.c** code, as follows:

```
dce_svc_set_progname("hello_program", &status);
```

The message printed by the program would, as a result, have looked like the following:

```
1994-04-05-20:13:34.500+00:00I----- hello_program \
    NOTICE hel main ...
Hello World
```

instead of looking like this:

```
1994-04-05-20:13:34.500+00:00I----- PID#9467 NOTICE hel
main ...
Hello World
```

where the default process ID information has been replaced by the string **hello_program** in the first example. The second example shows what the message looks like if the routine is not called. The **PID#** *nnnn* value is the value returned by **getpid()**.

This call is *optional*.

- **dce_svc_printf()**

  This is the normal call for writing or displaying serviceability messages. It cannot be called with a literal text argument; instead, the message text and other necessary information must be pre-specified in a file that is processed by the **sams** utility, which in turn outputs sets of tables from which the messages are extracted for output. The tutorial in "Simple Serviceability Interface Tutorial" on page 52 provides a brief example of how this is done.

  There are two main ways in which to call the routine. If a message has been defined in the **sams** file with both **sub-component** and **attributes** specified, then the **sams** output will include a convenience macro for the message that can be passed as the single argument to **dce_svc_printf()**, for example:

```
dce_svc_printf(HEL_S_HELLO_MSG);
```

  The convenience macro's name will be generated from the uppercase version of the message's **code** value (as specified in the **sams** file), with the string **_MSG** appended.

  If a convenience macro is not generated, or if you want to override some of the message's attributes at the time of output, then you must call the routine in its long form. For the **hel_s_hello** message, such a form of the call might look as follows:

```
dce_svc_printf(DCE_SVC(hel_svc_handle, ""), hel_s_main,\
 svc_c_sev_error | svc_c_route_stderr, hel_s_hello);
```

  DCE_SVC() is a macro that *must* be passed as the first argument to **dce_svc_printf()** if a convenience macro is not being used. It takes two arguments:

  – The caller's serviceability handle

– A format string for the message that is to be output

The format string is for use with messages that have been coded with argument specifiers. The **hel_s_hello** message had no argument specifiers, so an empty string is passed here to **DCE_SVC**. For an example of printing a message with arguments, see the end of this subsection.

The remaining arguments passed to **dce_svc_printf()** are as follows:

– Subcomponent table index

  This symbol was declared in the **sub-component** list coded in Part II of the **sams** file; its value is used to index into the subtable of messages in which the desired message is located.

– Message attribute(s)

  This argument consists of one or more attributes to be applied to the message that is to be printed. Note that you *must* specify at least a severity here (for a list of message severity values, see “Specifying Message Severity” on page 67). Multiple attributes are ORed together, as shown in the example.

  There are four categories of message attributes:

  routing
  The available routing attribute constants are

  - **svc_c_route_stderr**
  - **svc_c_route_nolog**

  However, most routing is done either by passing specially formatted strings to **dce_svc_routing()** or by environment variable values. See “How to Route Messages” on page 69 for more detailed information.

  severity
  The available severity attribute constants are

  - **svc_c_sev_fatal**
  - **svc_c_sev_error**
  - **svc_c_sev_warning**
  - **svc_c_sev_notice**
  - **svc_c_sev_notice_verbose**

  For more detailed information, see “Specifying Message Severity” on page 67.

  action
  The available message action attribute constants are

  - **svc_c_action_abort**
  - **svc_c_action_exit_bad**
  - **svc_c_action_exit_ok**
  - **svc_c_action_brief**

  For more detailed information, see “Message Action Attributes” on page 74.

  debug level
  Nine different debug levels can be specified. For more detailed information, see “Using Serviceability for Debug Messages” on page 78.

– message ID

This argument consists of the message’s **code**, as declared in the **sams** file.

As an example of how to use format specifiers in messages, consider the following **sams** file fragment, in which we define a second message for the **hello_svc.c** application:

```
start
code        hel_s_testmessage
text        "This message has exactly %d not %d argument(s)"
explanation     "This message is to show how to pass arguments"
action       "None required."
end
```

The message could be printed by a call like the following:

```
dce_svc_printf(DCE_SVC(hel_svc_handle, "%d%d"), hel_s_main,\
        svc_c_sev_notice | svc_c_route_stderr,\
        hel_s_testmessage, 2, 7);
```

Note the format specifiers passed in the format string to **DCE_SVC**, and the argument values passed at the end of the argument list. This call would cause the following message to be printed:

```
1994-04-06-20:06:33.113+00:00I----- hello \
     NOTICE hel main hello_svc.c line_nr 0xa444e208
This message has exactly 2 not 7 argument(s)
```

Extended Format Notation for Message Text

A slightly extended notation allows you to define message texts in the **sams** file that will (if desired) have format specifiers in their application code forms (that is, in the **.c** and **.msg** files output by **sams**), but which will be replaced by some specified string constant in the message texts that are generated for documentation use (that is, in the **.sml** and **.man** files).

The notation consists in surrounding the format specifier and alternative constant with < and > (angle bracket) characters, and separating the two with a **|** (vertical bar). (You can use a preceding \ (backslash) to escape these symbols.) For example, the following message text field:

```
text     Can't open input file %s for reading
```

would become something like the following:

```
text     Can't open input file <%s|filename> for reading
```

This message text definition, when processed by **sams**, would generate a format string with **%s** in the **.c** and message files, but this format specifier would be replaced by the string *filename* in the **.sml** and **.man** file versions.

Specifying Message Severity

Production (that is, nondebug) serviceability messages are categorized by their *severity level*, which implies various important things about the kind of situation that causes the message to be printed. Every message’s severity is stated in the text of the message itself (for example, **NOTICE** in the examples given previously shows that the messages are informational notices), and the serviceability routines can route and process messages differently on the basis of their severity levels.

Severity levels are attached to messages either when the messages are defined (in the **sams** file) or when the messages are written (by specifying an argument to the routine writing the message). These severity levels can then be used at runtime as the basis on which to route the messages (the way this is done will be explained in the next section).

Thus, each severity level is represented by a *constant* by which it is specified in program code, and a *name* by which it is referred to in routing files and environment variables. Each level's name and constant is shown, together with an explanation, in Table 2.

Table 2. Serviceability Message Severities

Name	Specifier	Meaning
FATAL	**svc_c_sev_fatal**	A fatal error has occurred; the program is about to exit.
ERROR	**svc_c_sev_error**	An error has occurred.
WARNING	**svc_c_sec_warning**	An error has been detected; the program is continuing execution.
NOTICE	**svc_c_sev_notice**	A nonerror event has occurred; this message is an informational notice of it.
NOTICE_VERBOSE	**svc_c_sev_notice_verbose**	A nonerror event has occurred; this message is a verbose informational notice of it.

Detailed explanations of the levels are as follows:

FATAL
: Fatal error exit: An unrecoverable error (such as database corruption) has occurred which will probably require manual intervention to be corrected. The program usually terminates immediately after such an error.

ERROR
: Error detected: An unexpected event that is nonterminal (such as a timeout), or is correctable by human intervention, has occurred. The program will continue operation, although some functions or services may no longer be available. This severity level may also be used to indicate that a particular request or action could not be completed.

WARNING
: Correctible error: An error occurred that was automatically corrected (for example, a configuration file was not found, and default values were used instead). This severity level may also be used to indicate a condition that *may* be an error if the effects are undesirable (for example, removing all files as a side effect of removing a nonempty directory), or to indicate a condition which, if not corrected, will eventually result in an error (for example, a printer's running low on paper).

NOTICE
: Informational notice: A significant routine major event has occurred; for example, a server has started.

NOTICE_VERBOSE
: Verbose information notice: A significant routine event has occurred; for example, a directory entry was removed.

Note that debug messages are identified as such by their own set of levels; see "Using Serviceability for Debug Messages" on page 78 for more information.

How to Route Messages

Serviceability messages can be written to any of the normal output destinations. Routing for serviceability messages can be specified in any of three different ways (in ascending order of precedence):

1. By the contents of a routing file
2. By the contents of a routing environment variable
3. By calling the **dce_svc_routing()** routine (often as part of processing an application's command-line arguments)

Additional routing (that is, in addition to whatever routing has been specified by the means described) of a message to standard error can be performed in either of the following two ways:

- By specifying the routing as one of the message's **attributes** (in the **sams** file definition of the message)
- By specifying the attribute in the call to **dce_svc_printf()** (or other serviceability output routine) to generate the message

Routing a message actually consists of specifying two things:

- How the message should be processed (the form it should be put in)
- Where the message should be sent (its destination)

The two specifications are sometimes closely interrelated, and sometimes specifying a certain destination implies that the message must be put into a certain form. This fact allows certain combinations of processing and destination to be abbreviated.

In the following sections, each of the ways to route serviceability messages is described.

Note that debug messages are routed by a similar, but slightly different, technique. For a full description, see "Using Serviceability for Debug Messages" on page 78.

Using a Routing File

If a file called *dce-local-path***/svc/routing** exists, the contents of the file (if in the proper format) will be used to determine the routing of messages written via serviceability routines.

The value of *dce-local-path* is usually **/opt/dcelocal**; the default location of the serviceability routing file is usually **/opt/dcelocal/svc/routing**. However, a different location for the file can be specified by setting the value of the environment variable **DCE_SVC_ROUTING_FILE** to the complete pathname.

The routing file consists of formatted strings specifying the routing desired for the various kinds of messages (based on message severity). Each string consists of three fields as follows:

```
sev:out_form:dest[;out_form:dest . . . ] [GOESTO:{sev | comp}]
```

where:

sev
: Specifies the severity level of the message, and must be one of the following:

- **FATAL**
- **ERROR**
- **WARNING**
- **NOTICE**
- **NOTICE_VERBOSE**

The meanings of these severity levels are explained in detail in “Specifying Message Severity” on page 67.

out_form
: (output form) Specifies how the messages of a given severity level should be processed, and must be one of the following:

BINFILE
: Write these messages as binary log entries.

TEXTFILE
: Write these messages as human-readable text.

FILE
: Equivalent to **TEXTFILE**.

DISCARD
: Do not record messages of this severity level.

STDOUT
: Write these messages as human-readable text to standard output.

STDERR
: Write these messages as human-readable text to standard error.

Files written as **BINFILE**s can be read and manipulated with a set of log file functions, or with the **svcdumplog** command. For further information, see “Logging and Log Reading” on page 73.

The *out_form* specifier may be followed by a two-number specifier of the form:

```
.gens.count
```

where:

gens
: Is an integer that specifies the number of files (that is, generations) that should be kept.

count
: Is an integer specifying how many entries (that is, messages) should be written to each file.

The multiple files are named by appending a dot to the simple specified name, followed by the current generation number. When the number of entries in a file reaches the maximum specified by *count*, the file is closed, the generation number is incremented, and the next file is opened. When the maximum generation number files have been created and filled, the generation number is reset to 1, and a new file with that number is created and written to (thus overwriting the already-existing file with the same name), and so on, as long as messages are being written. Thus the files wrap around to their beginning, and the total number of log files never exceeds *gens*, although messages continue to be written as long as the program continues writing them.

dest (destination) Specifies where the message should be sent and is a pathname. The field can be left blank if the *out_form* specified is **DISCARD**, **STDOUT**, or **STDERR**. The field can also contain a **%ld** string in the filename that, when the file is written, is replaced by the process ID of the program that wrote the messages. Filenames may *not* contain colons, semicolons, or periods.

Multiple routings for the same severity level can be specified by simply adding the additional desired routings as semicolon-separated strings in the following format:

```
out_form:dest
```

For example, consider the following:

```
FATAL:TEXTFILE:/dev/console
WARNING:DISCARD:--
NOTICE:BINFILE.50.100:/tmp/log%ld;STDERR:-
```

These strings specify that

- Fatal error messages should be sent to the console.
- Warnings should be discarded.
- Notices should be written both to standard error and as binary entries in files located in the **/tmp** directory. No more than 50 files should be written, and there should be no more than 100 messages written to each file. The files will have names of the form

  /tmp/log*process_id.nn*

  where *process_id* is the process ID of the program originating the messages, and *nn* is the generation number of the file.

The **GOESTO** specifier allows messages for the severity whose routing specification it appears in to be routed to the same destination (and in the same output form) as those for the other, specified, severity level (or component name). For example, the following specification:

```
WARNING:STDERR:;GOESTO:FATAL
FATAL:STDERR:;FILE:/tmp/foo
```

means that **WARNING** messages should show up in three places: twice to **stderr**, and then once to the file **/tmp/foo**.

Note that a **GOESTO** specification should be the last element in a multidestination route specification.

Routing by Environment Variable

Serviceability message routing can also be specified by the contents of certain environment variables. If environment variables are used, the routes they specify will override any conflicting routings specified by a routing file.

The routings are specified (on the basis of severity level) by putting the desired routing instructions in the following environment variables:

- **SVC_FATAL**
- **SVC_ERROR**
- **SVC_WARNING**
- **SVC_NOTICE**
- **SVC_NOTICE_VERBOSE**

Each variable should contain a single string in the following format:

out_form:*dest*;[*out_form*:*dest* . . .] [**GOESTO**:{*sev* | *comp*}]

where *out_form* and *dest* have the same meanings and form as described in “Using a Routing File” on page 69. Multiple routings can be specified with semicolon-separated additional strings specifying the additional routes, as shown.

Calling dce_svc_routing() to Set Routing

Message routing can be set up by the application itself, by calling the routine **dce_svc_routing()** and passing to it a string formatted in the same way as a line of text from a routing file. The routine must be called separately for each severity level. When routing is specified this way, the routings so specified will override any conflicting routings specified by environment variable or routing file (as described in the preceding sections). This is especially useful for setting routes from command-line arguments.

For example, to set routing in this way for the **hello_svc.c** application described previously, use the following code:

```
unsigned_char_t *my_route = "NOTICE:STDOUT:-;TEXTFILE:/tmp/my_log";
unsigned_char_t *error_route = "ERROR:TEXTFILE:/tmp/errors_%ld";

dce_svc_routing(my_route, &status);
if (status != svc_s_ok)
{
  printf("dce_svc_routing failed\n");
  exit(1);
}

dce_svc_routing(error_route, &status);
if (status != svc_s_ok)
{
  printf("dce_svc_routing failed\n");
  exit(1);
}
```

Additional Routing by Attribute

Limited additional routing for messages can be specified by attribute, either in the message definition itself in the **sams** file or as part of the argument list to **dce_svc_printf()**. Two routing attribute specifiers are available:

svc_c_route_stderr
Route the message to standard error.

svc_c_route_nolog
Discard the message.

Note also the **svc_c_action_brief** attribute, which is described in “Message Action Attributes” on page 74.

Table of Message Processing Specifiers

As was seen, message processing can be specified either by text strings (read from an environment variable or routing file, or passed to a routine) or, to a limited degree, by attribute in the message definition or when the message is output. Table 3 shows all the available types of serviceability message processing; the *name* by which it is specified in strings, and the *attribute* (where it exists) by which it is specified in message definitions and calls are both given, along with the meaning of each.

Table 3. Serviceability Message Processing Specifiers

Name	Attribute	Meaning
BINFILE		Write binary log entry.
TEXTFILE		Write human-readable text.
FILE		Equivalent to **TEXTFILE**.
DISCARD	**svc_c_route_nolog**	Do not record.
STDOUT		Write human-readable text to standard output.
STDERR	**svc_c_route_stderr**	Write human-readable text to standard error.
GOESTO		Route messages in same way as named level or component.

Logging and Log Reading

The serviceability interface includes a set of functions for reading and manipulating log files written as **BINFILE**s (see “Using a Routing File” on page 69).

dce_svc_log_open()
: Opens a log file for reading.

dce_svc_log_get()
: Reads the next entry from a log file. It returns the contents thereof in the form of a filled-in **prolog** structure to which it returns a pointer (see below for a description of the structure fields).

dce_svc_log_rewind()
: Returns log processing back to the first message in the log file.

dce_svc_log_close()
: Closes the open log file.

The contents of the log **prolog** structure (defined in **dce/svclog.h**) are as follows:

int *version*
: Version number of the interface that generated the message.

utc_t *t*
: Pointer to an opaque binary timestamp containing the time at which the message was written. The opaque timestamp can be converted to a **tm** structure by calling one of the DCE DTS **utc_** *xxx***()** routines.

unsigned32 *attributes*
: Message attributes, ORed together (a bit flag).

unsigned32 *message_index*
: Index number of message in message table (for example, **hel_s_hello** in the example at the beginning of this chapter).

pthread_t *thread_id*
: ID of application thread that caused the message to be written.

char **argtypes*
: The format-specifier string for the message.

int *argtypes_size*
: The number of format specifiers for the message.

char **fac_name*
: The component or subcomponent (“facility”) name string.

char **message_text*
: Message text string.

char **progname*
: Program name string, set by the application’s call to **dce_svc_set_progname()**.

char **file*
: Filename string identifying file from which entry was read.

int *line*
: Line number in *file* from where the message was printed.

int *file_size*
: Length of filename string.

Message Action Attributes

Routing and severity attributes affect what happens to the messages they are applied to, and nothing else. However, there is an additional set of attributes that, when applied to a message, mainly affect *what happens to the program* after the message is sent:

svc_c_action_abort
: Causes the program to abort (with core dump) as soon as the message is output.

svc_c_action_exit_bad
: Causes the program to exit (with failure status) as soon as the message is output.

svc_c_action_exit_ok
: Causes the program to exit (with successful status) as soon as the message is output.

svc_c_action_brief
: Suppresses the standard *prolog* of the message. The prolog of a serviceability includes all the nonmessage information that is output before the message text itself. The prologs of all messages can be suppressed by setting the **SVC_BRIEF** environment variable; see the next section.

Suppressing the Serviceability Message Prolog

You can suppress the prolog (nonmessage text) part of all serviceability messages generated by an application by setting the value of the **SVC_BRIEF** environment variable to 1.

The prolog of a serviceability consists of all the nonmessage information that is output before the message text itself. For example, examine the following message:

```
1994-04-05-20:13:34.500+00:00I----- PID#9467 \
     NOTICE hel main hello_svc.c  line_nr 0xa444e208
Hello World
```

In this example, the first line is the message prolog, and the second line is the message text. If the message were generated with the **SVC_BRIEF** environment variable set to 1, the message would appear as follows:

```
Hello World
```

Prologs of separate messages can be suppressed selectively through the use of the **svc_c_action_brief** attribute; see the previous section.

Serviceability Use of the __FILE__ Macro

Whenever a serviceability message is generated, information identifying the source file and line at which the invoked routine was called is included in the message information. This information appears in the text-form nonerror messages, and it is also written into the binary form serviceability logs (when binary logs are specified). The information also appears in the text form of messages announcing error situations. For example:

```
1994-07-20-11:11:09.906-04:00I----- sample_server FATAL \
     smp server sample_server.c 2851 0xa44b0c18
server_renew_identity(): login context has not been certified \
     (dce / sec)
```

(The preprocessor variable **DCE_SVC_WANT__FILE__** (in **dce/dce_svc.h**) will be defined or undefined depending on whether or not the serviceability component has been set up on your system to include the filename and line number information in serviceability messages.)

The serviceability routines receive the source file information from **DCE_SVC__FILE__**, which, by default, is defined to be the C preprocessor macro **__FILE__**. However, if you desire to avoid these macro expansions in your application code, you can redefine the symbol to be some kind of variable. For example:

```
#define DCE_SVC__FILE__ myfile
#include <dce/dce.h>
static char myfile[] = __FILE__;
```

Forcing Use of the In-Memory Message Table

As described elsewhere in this chapter, the **dce_msg_define_msg_table()** routine can be called by an application to initialize an in-memory copy of its message table, thus freeing the application from depending on its message catalog’s being properly installed for its serviceability messages to be properly generated.

However, the serviceability routines will still, by default, attempt first to retrieve a specified message from the message catalog, even if an in-memory table has been initialized; only if the message catalog cannot be found will the in-memory table be used.

You can change the default behavior of the serviceability routines by setting the **SVC_NOXPGCAT** environment variable to 1 (or any nonzero value). This will force the routines to always go to the in-memory table for the specified message; they will never look for the message catalog.

Dynamically Filtering Messages Before Output

The serviceability interface provides for a hook into the message-output mechanism that allows applications to decide at the time of messaging whether the given message should be output or not. The application defines its own routine to perform whatever checking is desired, and installs the routine with a call to **dce_svc_define_filter()**.

In addition, an application that installs such a message-filtering routine can also define and install a routine that can be called remotely to alter the operation of the filter routine. The remote-control routine is installed by the same call to **dce_svc_define_filter()**.

The two routines must have the following signatures. The yes/no routine you define and install is as follows:

```
boolean your_filter_routine(
                dce_svc_prolog_t prolog,
                va_list args)
```

The filter remote-control call is as follows:

```
void your_filter_remote_control(
                  idl_long_int arg_size;
                  idl_byte *arg;
                  error_status_t *status)
```

Once installed, the filter routine will be automatically invoked every time a serviceability routine is called to output a message. The filter receives a *prolog* argument that contains all the pertinent information about the message. If the filter returns TRUE, the message is output per the original serviceability call. If the filter returns FALSE, the message is not output. The information in the *prolog* allows such decisions to be made on the basis of severity level, subcomponent, message index, and so on. Its fields are as follows:

dce_svc_handle_t *handle*
: Serviceability handle of the application writing the message.

int *version*
: Version number of the interface that generated the message.

utc_t *t*
: Pointer to an opaque binary timestamp containing the time at which the message was written. The opaque timestamp can be converted to a **tm** structure by calling one of the DCE DTS **utc_...()** routines.

const char **argtypes*
: The format-specifier string for the message.

unsigned32 *table_index*
: **sams** file in “Defining the Message” on page 53.

unsigned32 *attributes*
: Message attributes, ORed together.

unsigned32 *message_index*
: Index number of the message in the message table (for example, **hel_s_hello** in the example at the beginning of this chapter).

char **format*
: Format argument values for the message.

const char **file*
: Filename string identifying the file to which the message is to be output.

char *progname[dce_svc_c_progname_buffsize]*
: Program name string, set by the application’s call to **dce_svc_set_progname()**.

int *line*
: Line number in *file* from where the message was printed.

pthread_t *thread_id*
: ID of the application thread that is causing the message to be output.

The filter remote control routine is part of the remote serviceability interface, which is described in detail in “Using the Remote Serviceability Interface” on page 84. Its operation is simple. If filter remote control is desired, the filter routine should be coded so that its operation can be switched to the various desired alternatives by the values of static variables to which it has access. These variables are also accessible to the remote control routine, and can be changed by it. The filter routine receives an argument string (which it uses to set the variables) whose contents are entirely application defined.

The following code fragments show a skeleton filter that can be added to the **hello_svc.c** example at the beginning of this chapter:

```
#include <stdarg.h>
#include <dce/svcfilter.h>

       <. . .>

/*****
* Filter routine-- once installed, this routine will be called
*        automatically every time a serviceability
*        routine (in our case, dce_svc_printf()) is
*        called to write a message.
*****/
boolean hel_filter(dce_svc_prolog_t  prolog,
          va_list      args)
{

  /* Code could be inserted here to test the values of static
    variables that would control the operation of the filter,
    and which could be altered by calling the filter control
    routine below. */

  printf("The progname is %s\n", prolog->progname);

  if (prolog->attributes | svc_c_sev_notice)
    printf("This is a Notice-type message\n");
  switch (prolog->table_index)
  {
    case hel_s_main:
      printf("Main subcomponent\n");
      break;
    default:
      printf("Error\n");
      break;
  }


  /* The routine returns 1, thus permitting the output
    operation to go ahead; if 0 were returned here, the
    operation would be suppressed ... */

  return 1;

}

/*****
* Filter Control routine-- this routine is normally called
*            through the remote interface.
*****/
void hel_filter_control(idl_long_int arg_size,
            idl_byte *arg,
            error_status_t *status)
{

  /* Code would be inserted here to interpret the arg passed
    and, on the basis of that, change the value(s) of one
    or more static variables that control the operation of
    hel_filter()                     */

}
/*****
* install_filters-- calls dce_svc_define_filter() to install
*         the above 2 routines. Note that this must
*         be done after dce_svc_register() is
*   called, not before.
*****/
void install_filters()
{
  unsigned32 status;

  dce_svc_define_filter(hel_svc_handle, hel_filter, \
   hel_filter_control, &status);
}
```

Using Serviceability for Debug Messages

Apart from the **dce_svc_printf()** routine for writing production serviceability messages, the interface provides several macros that can be used for debug messaging in a server. The advantages in using these macros in debugging are the following:

- All of the debug messaging code can easily be compiled in or out of the executable by changing the value of a compilation switch.
- Nine levels of debug messaging are provided for; the active level of debug messaging can be controlled through the remote serviceability interface or by a value passed to the server at startup.
- One of the macros allows message text to be specified in the call itself, rather than extracting it by message ID from the message table.

The debug serviceability messaging routines are the following:

- **DCE_SVC_LOG()**

Outputs a message specified by the message ID. The main differences between using this routine and using **dce_svc_printf()** to write a message are (1) that **DCE_SVC_LOG()** generates records *only* in binary format, and (2) the macro can be compiled out of the executable by turning off debugging.

Suppose the following message had been defined in the **hel.sams** file for the example application at the beginning of this chapter:

```
start
code        hel_s_debug_message_1
subcomponent    hel_s_main
attributes     "svc_c_debug3 | svc_c_route_stderr"
text        "This is a level 1 test debug message"
explanation     "Debug level 3 test"
action       "None required."
end
```

The following call in **hello_svc.c** would have written this message as a binary record to the specified route, provided that debug level 3 had been activated:

```
DCE_SVC_LOG((HEL_S_DEBUG_MESSAGE_1_MSG));
```

Note the use of the double parentheses. This is made necessary by the fact that it is a macro that takes a variable number of arguments. Note also the use of the convenience macro form of the message. A full form of the call, with all arguments explicitly specified, would have been as follows:

```
DCE_SVC_LOG((DCE_SVC(hel_svc_handle, ""), \
   hel_s_main, svc_c_debug3, hel_s_debug_message_1));
/*    |       |         |        */
/* table_index   |         |        */
/*       debug level      |        */
/*               message ID
*/
```

Debug messages, like normal serviceability messages, can also contain format specifiers and argument lists.

- **DCE_SVC_DEBUG()**

  Outputs a message whose text is specified in the call. For example, the following call could have appeared in **hello_svc.c**:

```
DCE_SVC_DEBUG((hel_svc_handle, \
/*         |                  */
/*       handle                  */
/*                            */
  hel_s_main, svc_c_debug2, "A Debug Level %d message", 2));
/*    |       |         |         | */
/* table_index    |         |         | */
/*      debug level       |         | */
/*               message text      | */
/*                       argument
*/
```

  Note here too the use of the double parentheses.

  Note also that **DCE_SVC_DEBUG** cannot be used with the convenience macro forms of serviceability messages.

- **DCE_SVC_DEBUG_ATLEAST()**

  Tests the active debug level for a subcomponent. Returns TRUE if the debug level (set by calling **dce_svc_debug_set_levels()**; see "Setting Debug Levels" on page 80 on page 80) is not less than the specified level; otherwise returns FALSE. For example, the following call would return TRUE if the debug level for the **hel_s_main** subcomponent of the **hello_svc** application had been set to **svc_c_debug2** or any higher value:

```
DCE_SVC_DEBUG_ATLEAST(hel_svc_handle, hel_s_main, svc_c_debug2);
```

This macro can be used to test the active debug level and avoid calling a debug output routine if the level of its message is disabled at the time of the call (disabling the level does not stop any routines from being executed; it only suppresses the output messages at that level). See “Performance Costs of Serviceability Debugging” on page 83 for more information.

- **DCE_SVC_DEBUG_IS()**

  Tests the active debug level for a subcomponent. Returns TRUE if the debug level is the same as that specified in the call; otherwise returns FALSE. For example, the following call would return TRUE only if the debug level for **hel_s_main** had been set to **svc_c_debug2**:

  ```
  DCE_SVC_DEBUG_IS(hel_svc_handle, hel_s_main, svc_c_debug2);
  ```
- **dce_assert()**

  Evaluates an **int** expression passed to it and, if the expression evaluates to 0 (that is, if the expression is false), automatically calls **dce_svc_printf()** with parameters that will cause a message with a severity level of **svc_c_sev_fatal** (that is, fatal) and an action attribute of **svc_c_action_abort** to be printed that will identify the following:
  - The expression
  - The source file in which the assertion failed
  - The line at which the assertion failed

  For example, the following call will cause the failed expression (namely, the string) to be printed and the program to be aborted.

  ```
  dce_assert(hel_svc_handle, ("Test diagnostic message" == NULL))
  ```

  A NULL can be substituted for the serviceability handle as the first argument.

It is very important that debug messages not be used for errors that can occur during ordinary operation. This is because the debug messaging code can be omitted when compiling for production.

Setting Debug Levels

Nine serviceability debug message levels are available. The precise meaning of each level for an application is left to the developer; but the general intention is that ascending to a higher level (for example, from **svc_c_debug2** to **svc_c_debug3**) should increase the level of information detail.

Setting debug messaging at a certain level means that all levels up to and including the specified level are enabled. For example, if the debug level is set at **svc_c_debug4**, then the **svc_c_debug1**, **svc_c_debug2**, and **svc_c_debug3** levels are enabled as well.

A message can have a debug level attached to it in either of two ways:

- The debug level can be specified as one of the **attributes** in the message's definition in the **sams** file.

- If **DCE_SVC_DEBUG()** or **DCE_SVC_LOG()** is used to output the message, the debug level is specified in the call.

The debug level can be set by calling **dce_svc_debug_set_levels()** and passing to it a specially formatted string (the debug level is also set when debug routing is specified; see the next section for further information). Levels can be separately specified for subcomponents. For example, suppose two subcomponents (rather than one) had been defined in the **sams** file for the **hello_svc** application at the beginning of this chapter, as follows:

```
# Part II
serviceability table hel_svc_table handle hel_svc_handle
start
  subcomponent  hel_s_main   "main" hel_i_svc_main
  subcomponent  hel_s_utils   "utils" hel_i_svc_utils
end
```

The following string would, when passed to **dce_svc_debug_set_levels()**, set the debug level for the **main** subcomponent to be **svc_c_debug1**, and the debug level for the **utils** subcomponent to be **svc_c_debug4**:

```
unsigned_char_t *levels = "hel:main.1,utils.4";
```

The general format for the debug level specifier string is as follows:

```
component:sub_comp.level,sub_comp.level,. . .
```

where:

component
: Is the three-character component code for the program.

*sub_comp***.** *level*
: Is a subcomponent name, followed (after a dot) by a debug level (expressed as a single digit from 1 to 9). Note that multiple subcomponent/level pairs can be specified in the string.

If there are multiple subcomponents, and it is desired to set the debug level to be the same for all of them, then the following form will do this (where the ***** (asterisk) specifies all subcomponents).

```
component:*.level
```

The string can be passed to **dce_svc_debug_set_levels()** as follows:

```
dce_svc_debug_set_levels(levels, &status);
```

where **levels** is a string declared similarly to the example shown earlier in this section.

The nine serviceability debug message level specifiers are as follows:
- **svc_c_debug1**
- **svc_c_debug2**
- **svc_c_debug3**
- **svc_c_debug4**
- **svc_c_debug5**

- **svc_c_debug6**
- **svc_c_debug7**
- **svc_c_debug8**
- **svc_c_debug9**

Routing Debug Messages

Routing for serviceability debug messages can be specified in any of four ways:

- By calling the **dce_svc_debug_routing()** routine
- By the contents of the **SVC_** *CMP***_DBG** environment variable (where *CMP* is the three-character serviceability name of the component, in uppercase)
- By the contents of the routing file *dce-local-path***/svc/routing**
- By one of the message's **attributes** (as coded in the **sams** file)

In all but the last method, the routing is specified by the contents of a specially formatted string that is either included in the value of the environment variable, is part of the contents of the routing file, or is passed to the **dce_svc_debug_routing()** routine.

The general format for the debug routing specifier string is

component:*sub_comp*.*level*, ...:*out_form*:*dest* [*out_form*:*dest* ...] [**GOESTO**:{*sev* | *comp*}]

where:

component
: Specifies the component name.

sub_comp. *level*
: Specifies a subcomponent name, followed (after a dot) by a debug level (expressed as a single digit from 1 to 9). Note that multiple subcomponent/level pairs can be specified in the string.

The meanings of the remaining elements of the string are the same as those for the identically named elements in "How to Route Messages" on page 69.

Multiple routings for the same group of subcomponents can be specified by adding semicolon-separated strings of the following format:

out_form:*dest*

to the specification, in a form analogous to that followed for specifying production (nondebug) message routes, shown previously.

The following string would, when passed to **dce_svc_debug_routing()**, set the debug level and routing for all **hel** subcomponents:

```
unsigned_char_t *debug_routes = \
 "hel:*.4:TEXTFILE:/tmp/hel_debug_log_%ld;STDERR:-";
```

A debug level of **svc_c_debug4** is specified, and all debug messages of that level or lower will be written both to standard error and in text form to the following file:

/tmp/hel_debug_log_*process_ID*

where *process_ID* is the process ID of the program writing the messages.

The specification string could be passed to **dce_svc_debug_routing()** as follows:

```
dce_svc_debug_routing(debug_routes, &status);
```

To specify the same routing by environment variable, the string following value should be assigned to **SVC_** *CMP***_DBG**:

```
hel:*.4:TEXTFILE:/tmp/hel_debug_log_%ld;STDERR:-
```

The same string information could also be inserted into the **SVC_** *CMP* environment variable or into the contents of the routing file.

Debug routing by attribute (as specified in the **sams** file) is done in the same way as routing for normal messages. See “Additional Routing by Attribute” on page 72.

Performance Costs of Serviceability Debugging

If serviceability debugging routines are used in an application, one of three different things can happen to any given debugging routine at runtime:

- The routine is called, and its output is generated (because the debug level associated with the message has been enabled).
- The routine is called, but its output is not generated (because the debug level associated with the message has been disabled).
- The routine call is not present in the application code because serviceability debugging has been compiled out (**DCE_DEBUG** was not defined when the application was compiled).

Note that, even if a certain debug level has been disabled, any routine or macro call to output a message with that level will still be executed unless other steps are taken to prevent this. The performance cost associated with such smothered calls will usually be insignificant, but situations can occur in which this will not be so.

For example, developers should understand the implications of supplying function calls as arguments to serviceability debug output routines (such as **DCE_SVC_DEBUG**). If the debug code is compiled in (that is, if **DCE_DEBUG** is defined), then the parameterized function calls will always be executed because the output routine itself will still be called—even though it will produce no output.

In situations like this, the desirable course of action is simply to not call the output routine at all if the currently set debug level has turned it into a no-op. This can be done by using the **DCE_SVC_DEBUG_ATLEAST** macro to check the current level, as shown in the following example:

```
if (DCE_SVC_DEBUG_ATLEAST(hel_svc_handle, hel_s_main, vc_c_debug3))
{
  DCE_SVC_DEBUG((
    hel_svc_handle,
    hel_s_main,
    svc_c_debug3,
    " a_function_call() return value is: %s",
     a_function_call(parm, status)));
}
```

The normal performance cost of a serviceability logging operation normally amounts to one mutex lock and (usually) one file lock access per operation.

Using the Remote Serviceability Interface

Serviceability is primarily a mechanism intended to be used by servers. Like other server functionality, it should be remotely controllable by properly authorized entities. This allows such things as message routing and debug levels to be adjusted without having to restart the server.

The standard remote serviceability interface is defined in the file **/usr/include/dce/service.idl**.

An application server using serviceability is responsible for providing routines that implement the operations defined in **service.idl**. However, implementing the operations themselves is a simple matter of calling library routines that actually perform them. The job of the application implementation is mainly to check the authorization of the remote caller and then either reject the request (if authorization is found to be insufficient) or call the appropriate library routine to perform the operation.

Table 4 lists such remote operations.

Table 4. Remote Operations by Application Servers

Server Implementation	Library Routine	Purpose
*com*_**svc_set_route()**	**dce_svc_routing()**	Remotely sets serviceability message routing.
*com*_**svc_set_dbg_route()**	**dce_svc_debug_routing()**	Remotely sets serviceability debug message routing.
*com*_**svc_set_dbg_levels()**	**dce_svc_debug_set_levels()**	Remotely sets serviceability debug message levels.
*com*_**svc_inq_components()**	**dce_svc_components()**	Returns a listing of all components that have been registered with the **dce_svc_register()** routine.
*com*_**svc_inq_table()**	**dce_svc_table()**	Returns the message table registered with a given component.
*com*_**svc_inq_routings()**	**dce_svc_routings()**	Returns a list of routings in effect for a component.
*com*_**svc_filter_control()**	**dce_svc_filter()**	Remotely controls the behavior of the serviceability message filtering routine (if one exists).
*com*_**svc_inq_stats()**	**dce_svc_inq_stats()**	Returns operating statistics.

Basic Steps in Setting Up the Remote Interface

To make the interface available, the developer must do the following:

1. Coding steps
   - Define the server implementation routines for the remote operations.
   - Initialize the serviceability interface manager entry point vector (manager EPV) with the implementation routines.
2. Build steps
   - Process the **service.idl** file to produce the following:

- Client stub

  This will be linked into the client object. The client itself can contain calls to the remote routines, expressed by their interface names.
- Server stub

  This will be linked into the server object (just as its own stub(s) are) to produce the server executable. Note that the server stub is generated with the **-no_mepv** IDL option, which allows the implementation routines to be named anything that suits the developer. This is why the EPV must be explicitly initialized with the implementation routines' addresses.

3. Runtime steps
   - At server startup:

     The binding handles that the server receives from the RPC runtime, and which it then registers both with the Name Server Interface (NSI) and the endpoint mapper under its own interface, must also be registered to the endpoint mapper with the serviceability interface. Note that servers *do not* explicitly register the serviceability interface with the NSI. Instead, they use their existing namespace entries without change. They *do* register the serviceability interface with their endpoint mapper.
   - For a client application:

     To call one of a server's remote serviceability routines, the client must import a binding handle using a NULL UUID; this operation will yield a plain handle. The client can then pass this handle to the desired remote serviceability routine and make the call. The server's host endpoint mapper will recognize the incoming serviceability UUID in the RPC, and will send the RPC on to one of the registered endpoints.

The following code fragments illustrate how to define, export, and access the serviceability remote interface.

Implementing the Remote Routines

The following code fragments show in skeletal form how an application's remote serviceability routines should be implemented. The pseudo-code references to access tests are calls to the application's ACL manager to assess the caller's authorization. For information on implementing an ACL manager, see the security chapters of the *OSF DCE Application Development Guide—Introduction and Style Guide* and the *OSF DCE Administration Guide—Core Components*.

```
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include <dce/dcesvcmsg.h>
#include <dce/svcremote.h>

struct serviceability_v1_0_epv_t    dce_svc_epv;

/*****
*
* hel_svc_set_route -- remote call-in to set routing.
*
*****/
static void
hel_svc_set_route(
  handle_t        h,
  idl_byte        where[],
  error_status_t     *st
)
{
  if (! your_test_write_access(h))
    *st = no_authorization_error;
  else
    dce_svc_routing(where, st);

}

/*****
*
* hel_svc_set_dbg_route -- remote call-in to set debug routing.
*
*****/
static void
hel_svc_set_dbg_route(
  handle_t        h,
  idl_byte        where[],
  error_status_t     *st
)
{

  if (! your_test_write_access(h))
    *st = no_authorization_error;
  else
    dce_svc_debug_routing(where, st);

}

       <. . .>

/*****
*
* hel_svc_inq_stats -- remote request for operating statistics.
*
*****/
static void
hel_svc_inq_stats(
  handle_t        h,
  dce_svc_stats_t     *stats,
  error_status_t     *st
)
{

  if (! your_test_access(h))
    *st = no_authorization_error;
  else
    /* operation is currently not implemented in library... */
    *st = svc_s_no_stats;

}

/*                              */
/* The table of slots is created by IDL from the service.idl  */
/* file, src/dce/utils/svc/service.idl, the output of which   */
/* is service.h. It's then the job of the application that   */
/* wishes to offer the remote operations to fill in the table  */
/* with the implementations' entry points. That's what's being */
/* done below. Typically the application simply interposes an */
/* appropriate ACL check between the entry into an       */
/* implementation and the subsequent call to the "real"     */
/* operation as implemented in the serviceability library.   */
/*                              */

serviceability_v1_0_epv_t dce_svc_epv = {
  hel_svc_set_route,
  hel_svc_set_dbg_route,
  hel_svc_set_dbg_levels,
  hel_svc_inq_components,
  hel_svc_inq_table,
  hel_svc_inq_routings,
  hel_svc_filter_ctl,
  hel_svc_inq_stats
};
```

Registering and Exporting the Remote Interface

The following code fragments show how the remote serviceability interface could be exported and registered by a **hello_svc** server. Note that only the steps that are closely or directly related to exporting and registering the server's and the serviceability remote interface are shown. For a full example of how to get a DCE server application up and running, see the *OSF DCE Application Development Guide—Introduction and Style Guide*.

The steps shown are the following:

1. Register interfaces with the RPC runtime
2. Request binding handles for the server interface from the RPC runtime
3. Request binding handles for the serviceabilty interface from the RPC runtime
4. Register both sets of binding handles with the endpoint map
5. Export both sets of binding handles to the namespace

Note that (for brevity's sake) status return checks have been omitted from this code.

```
 <. . .>

/* Register server interface/type_uuid/epv associations    */
/* with rpc runtime.                          */
rpc_server_register_if(timop_v1_0_s_ifspec, &type_uuid,
  (rpc_mgr_epv_t)&manager_epv, &status);

/* Register serviceability remote interface with rpc      */
/* runtime ...                             */
rpc_server_register_if(serviceability_v1_0_s_ifspec, &type_uuid,
  (rpc_mgr_epv_t)&dce_svc_epv, &status);

         <. . .>

/* Tell rpc runtime we want to use all supported protocol   */
/* sequences.                             */
rpc_server_use_all_protseqs(MAX_CONC_CALLS_PROTSEQ, &status);

/* Get server binding handles ...               */
rpc_server_inq_bindings(&hello_bind_vector_p, &status);

/* Get binding handles for serviceability remote       */
/* interface ...                           */
rpc_server_inq_bindings(&svc_bind_vector_p, &status);

         <. . .>/* Register endpoints with server interface ...         */
rpc_ep_register(hello_v1_0_s_ifspec, hello_bind_vector_p,
  (uuid_vector_t *)&obj_uuid_vec,
  (unsigned_char_t *)"hello server, version 1.0",
  &status);

/* Register endpoints with serviceability interface ...     */
rpc_ep_register(serviceability_v1_0_s_ifspec, svc_bind_vector_p,
  (uuid_vector_t *)&obj_uuid_vec,
  (unsigned_char_t *)"Hello SVC",
  &status);

/* Export server interface binding info to the namespace.    */
```

```
rpc_ns_binding_export(rpc_c_ns_syntax_dce, server_name,
  hello_v1_0_s_ifspec, hello_bind_vector_p,
  (uuid_vector_t *)&obj_uuid_vec, &status);
```

Importing and Accessing the Remote Interface

The following code fragments are intended to give an idea how a client might import both the **hello_svc** server's interface and its exported serviceability interface.

Note that (for brevity's sake) status return checks have been omitted from this code.

```
/* Import binding info from namespace. */
for (server_num = 0; server_num < nservers; server_num++)
{
  /* Begin the binding import loop. */
  rpc_ns_binding_import_begin(rpc_c_ns_syntax_dce,
    server_name[server_num], hello_v1_0_c_ifspec,
    &obj_uuid, &import_context, &status);

  /* Begin the svc binding import loop. */
  rpc_ns_binding_import_begin(rpc_c_ns_syntax_dce,
    server_name[server_num], NULL,
    &obj_uuid, &svc_import_context, &status);
  /* Import bindings one at a time. */
  while (1)
  {
    rpc_ns_binding_import_next(import_context,
      &bind_handle[server_num], &status);

    rpc_ns_binding_import_next(svc_import_context,
      &svc_bind_handle[server_num], &status);

    /* Select, say, the first binding over UDP. */
    rpc_binding_to_string_binding(bind_handle[server_num],
      &string_binding, &status);

    rpc_binding_to_string_binding(svc_bind_handle[server_num],
      &svc_string_binding, &status);

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

    rpc_string_binding_parse(svc_string_binding, NULL,
      &svc_protseq, NULL, NULL, NULL, &status);

    rpc_string_free(&string_binding, &status);
    ret = strcmp((char *)protseq, "ncadg_ip_udp");
    rpc_string_free(&protseq, &status);

    rpc_string_free(&svc_string_binding, &status);
    svc_ret = strcmp((char *)svc_protseq, "ncadg_ip_udp");
    rpc_string_free(&svc_protseq, &status);

    if ((svc_ret == 0) || (ret == 0))
    {
      break;
    }

  }

  /* End the binding import loop. */
  rpc_ns_binding_import_done(&import_context, &status);
  rpc_ns_binding_import_done(&svc_import_context, &status);

}
/* 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);

for (server_num = 0; server_num < nservers; server_num += 1)
  rpc_binding_set_auth_info(svc_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);
```

Chapter 5. The DCE Backing Store

This chapter describes the backing store library that DCE provides for the convenience of programmers who are writing DCE servers. A backing store is a persistent database or persistent object store from which typed data can be stored and retrieved by a key.

Note: Sometimes the backing store is called a database. For instance, the associated IDL file is **dce/database.idl**, and the name of the backing store routines begin with **dce_db_**. The backing store is, however, not a full-fledged database in the conventional sense, and it has no support for SQL or for any other query system.

Servers generally need to manage several objects. Good design often requires that the state of the objects be maintained over sequential instances of a particular server. For example, the ACLs used by a server should not need to be recalculated each time the system is rebooted. The backing store interface provides a way to store, into a file, any data that can be described with IDL so that it can persist across instances of software that run from time to time. For example, the ACL library uses the backing store library. The backing store routines can be used in servers, in clients or in standalone programs that do not involve remote procedure calls (RPCs). Backing store data should not be used for sharing data between processes.

Data in a Backing Store

The backing store interface provides the applications programmer with the capability for tagged storage and retrieval of typed data. The tag (or retrieval key) can be either a UUID or a standard C string. For a specific backing store, the data type must be specified at compile time and is established through the IDL encoding services. Each backing store can contain only a single data type.

Each data item (which may also be called a data object, or a data record) consists of the data stored in a single call to a storage routine. The storage routines are **dce_db_store()**, **dce_db_store_by_name()**, and **dce_db_store_by_uuid()**. Optionally, data items may have standard headers. If a backing store has been created to use headers, then every data item has the header.

A program can have more than one backing store open at the same time.

Using a Backing Store

Although the backing store library is a generalized service, you are encouraged to use it in a particular, standardized way. You should use the header and the recommended IDL interface format that are described in the following sections. Standardized use will ease the transition to later developments in DCE.

Header for Data

An optional standard header is available for data objects or items in the backing store. If it is employed, then the backing store library automatically maintains the **created**, **modified**, and **modified_count** fields, as shown in the following IDL description, taken from the **dce/database.idl** file:

```
/* The standard header for each "object" n the database. */

typedef struct dce_db_dataheader_s_t {
  uuid_t      uuid;
  uuid_t      owner_id;
  uuid_t      group_id;
  uuid_t      acl_uuid;
  uuid_t      def_object_acl;
  uuid_t      def_container_acl;
  unsigned32    ref_count;
  /* The following fields are updated by the library */
  utc_t       created;
  utc_t       modified;
  unsigned32    modified_count;
} dce_db_dataheader_t;

typedef enum {
  dce_db_header_std,
  dce_db_header_acl_uuid,
  dce_db_header_none
} dce_db_header_type_t;

typedef union switch (dce_db_header_type_t type) tagged_union {
  case dce_db_header_none:      /* none */ ;
  case dce_db_header_std:       dce_db_dataheader_t h;
  case dce_db_header_acl_uuid:    uuid_t acl_uuid;
} dce_db_header_t;

void dce_db_header_convert(
  [in]    handle_t       h,
  [in,out]  dce_db_header_t     *data,
  [out]    error_status_t     *st
);
```

The **acl_uuid** field is intended for use as a UUID retrieval key in a server's ACL database.

The User Interface

The recommended, standardized backing store IDL interface for a server looks like the following, where *XXX* is the server name:

```
interface XXX_convert
{
  import "dce/database.idl"

  typedef XXX_data_s_t {
    dce_db_header_t header;  /* Header must be first! */
    /* (server-specific data goes here) */
  } XXX_data_t;

  void XXX_data_convert(
    [in]    handle_t     h,
    [in, out]  XXX_data_t    *data
    [out]    error_status_t  *st
  );
}
```

It should be compiled with the following Attribute Configuration File (ACF), which instructs the **idl** compiler to write the data conversion routine into the *XXX*_**cstub.c** file:

```
interface XXX
{
  [encode, decode] XXX_data_convert([comm_status] st);
}
```

The IDL Encoding Services

When remote procedure call sends data between a client and a server, it serializes the user's data structures by using the IDL encoding services, described in "Chapter 16. Writing Internationalized RPC Applications" on page 281 of this book.

Encoding and Decoding in the Backing Store

The backing store uses this same serialization scheme for encoding and decoding, informally called *pickling*, when storing data structures to disk. The IDL compiler, **idl**, writes the routine that encodes and decodes the data. This routine is passed to **dce_db_open()**, remembered in the handle, and used by the following store and fetch routines:

- **dce_db_fetch()**
- **dce_db_fetch_by_name()**
- **dce_db_fetch_by_uuid()**
- **dce_db_header_fetch()**
- **dce_db_store()**
- **dce_db_store_by_name()**
- **dce_db_store_by_uuid()**

Conformant Arrays Not Allowed

You cannot use conformant arrays in objects stored to a backing store. This is because the IDL-generated code that encodes (pickles) the structure has no way to predict or detect the size of the array. When the object is fetched, there will likely be insufficient space provided for the structure, and the array's data will destroy whatever is in memory after the structure.

To illustrate the problem more clearly, here is an example. An IDL file has a conformant array, **na**, as an object in a **struct**:

```
typedef struct {
  unsigned32 length;
  [size_is(length)]
  unsigned32 numbers[];
} num_array_t
typedef struct {
  char     *name;
  num_array_t  na;
} my_type_t;
```

The **idl** compiler turns the IDL specification into the following **.h** file contents:

```
typedef struct {
  unsigned32 length;
  unsigned32 numbers[1];
} num_array_t
```

```
typedef struct {
  idl_char    *name;
  num_array_t   na;
} my_type_t;
```

When the object is fetched, and the array length is greater than the 1 (one) assumed in the **.h** file, the decoding operation destroys whatever follows **my_struct** in memory:

```
my_type_t my_struct;
dce_db_fetch(dbh, key, &my_struct, &st);
```

The correct method is to use a pointer to the array, not the array itself, in the IDL file. For example:

```
typedef struct {
  char      *name;
  num_array_t   *na;
} my_type_t;
```

The Backing Store Routines

Many of the backing store routines appear in three versions: plain, by name, and by UUID. The plain version will work with backing stores that were created to be indexed either by name, or by UUID; the restricted versions accept only the matching type. It is advantageous to use the restricted versions when they are appropriate because they provide type checking by the compiler, as well as visual clarity of purpose.

The backing store operations described in the following sections are supported.

Opening a Backing Store

The **dce_db_open()** routine creates a new backing store or opens an existing one. The backing store is identified by a filename. There are flags to permit the following choices:

- Create a new backing store or open an existing one.
- Create a new backing store indexed by name or UUID. (The choice depends upon the server's purpose.) This index is called the backing store key.
- Open an existing backing store read/write or read-only.
- Use the standard header or not.

Every backing store is created with one of the two possible index schemes, by name or by UUID, and you cannot subsequently open it for use with the other scheme. Also, once a backing store has been created with (or without) standard headers, you cannot subsequently open it the other way.

The routine returns a handle by which subsequent operations identify the backing store.

The following conventions for filenames are recommended:

*xxx***.acl**	ACL storage.
*xxx***.db**	Backing store filename.

Closing a Backing Store

The **dce_db_close()** routine frees the handle. It closes any open files and releases all other resources associated with the backing store.

Storing or Retrieving Data

The following routines store data into a backing store:

dce_db_store()

This routine can store into a backing store that is indexed by name or by UUID. The key’s type must match the flag that was used in **dce_db_open()**.

dce_db_store_by_name()

This routine can store only into a backing store that is indexed by name.

dce_db_store_by_uuid()

This routine can store only into a backing store that is indexed by UUID.

To retrieve data from a backing store, use the appropriate one of the following routines:

dce_db_fetch()

This routine can retrieve data from a backing store that is indexed by name or by UUID. The key’s type must match the flag that was used in **dce_db_open()**.

dce_db_fetch_by_name()

This routine can retrieve data only from a backing store that is indexed by name.

dce_db_fetch_by_uuid()

This routine can retrieve data only from a backing store that is indexed by UUID.

When storing or retrieving data, a function that was specified at open time converts between native format and on-disk (serialized) format. This function is generated from the IDL file by the IDL compiler.

Freeing Data

When fetching data, the encoding services allocate memory for the data structures that are returned. These services accept a structure and use **rpc_sm_allocate()** to provide additional memory needed to hold the data.

The backing store library does not know what memory has been allocated and, therefore, cannot free it. For fetch calls that are made from a server stub, this is not a problem because the memory is freed automatically when the server call terminates. For fetch calls that are made from a nonserver, the programmer is responsible for freeing the memory.

Programs that call the fetch or store routines, such as **dce_db_fetch()**, outside of a server operation (for instance, if a server does some backing store initialization, or in a standalone program) must call **rpc_sm_enable_allocate()** first.

Making or Retrieving Headers

The **dce_db_std_header_init()** routine initializes a standard backing store header from the values the caller provides in its arguments. It places the values into the header only and does not write into the backing store file. The **dce_db_header_fetch()** routine retrieves the header of an object in the backing store.

Performing Iteration

The following routines iteratively traverse all of the keys (name or UUID) in a backing store. The order of retrieval of the keys is indeterminate; they are not sorted, nor are they necessarily returned in the order in which they were originally stored. It is strongly recommended to use the locking and unlocking routines, **dce_db_lock()** and **dce_db_unlock()**, whenever performing iteration.

dce_db_iter_start()

This routine prepares for the start of iteration.

dce_db_iter_next()

This routine returns the key for the next item from a backing store that is indexed by name or by UUID. The **db_s_no_more** status code indicates that there are no more items.

dce_db_iter_next_by_name()

This routine returns the key for the next item only from a backing store that is indexed by name. Again, **db_s_no_more** indicates that no items remain.

dce_db_iter_next_by_uuid()

This routine returns the key for the next item only from a backing store that is indexed by UUID. Again, **db_s_no_more** indicates that no items remain.

dce_db_iter_done()

This routine is counterpart to **dce_db_iter_start()** and should be called when iteration is done.

dce_db_inq_count()

This routine returns the number of items in a backing store.

Deleting Items from a Backing Store

The following routines delete an item from a backing store.

dce_db_delete()

This routine deletes an item from a backing store that is indexed by name or by UUID. The key's type must match the flag that was used in **dce_db_open()**.

dce_db_delete_by_name()

This routine deletes an item only from a backing store that is indexed by name.

dce_db_delete_by_uuid()

This routine deletes an item only from a backing store that is indexed by UUID.

To delete an entire backing store, ensure that the data file is not open, and remove it. There is only one file.

Locking and Unlocking a Backing Store

The **dce_db_lock()** and **dce_db_unlock()** routines lock and unlock a backing store. If a backing store is already locked, **dce_db_lock()** provides an indication. A lock is associated with an open backing store's handle. The storage routines, **dce_db_store()**, **dce_db_store_by_name()**, and **dce_db_store_by_uuid()**, all acquire the lock before updating. Explicit use of locking is appropriate in some circumstances; for example, when reading or writing pairs (or multiples) of closely associated items in a backing store, or when using iteration.

The locks are advisory. It is possible to write a backing store even if it is locked so, if you want to rely upon the locks, you must always check them.

Example of Backing Store Use

For a full example of backing store use, see the *OSF DCE Application Development Guide—Introduction and Style Guide*.

The following brief example shows a portion of a server that manages an office telephone directory. Following are the relevant structures, defined in an IDL file:

```
typedef struct phone_record_s_t {
  [string,ptr] char    *name;
  [string,ptr] char    *email;
  [string,ptr] char    *phone;
  [string,ptr] char    *office;
} phone_record_t;

typedef struct phone_record_array_s_t {
              unsigned32   count;
  [ptr,size_is(count)]  phone_record_t *entry;
} phone_record_array_t;

typedef struct phone_data_s_t {
  dce_db_header_t h;
  phone_record_t ph;
} phone_data_t;
/*
 * The following routine returns the entire contents of the
 * directory from the backing store by using the iteration
 * routines. First, the portion of the IDL file that
 * defines the routine's RPC format:
 */
[idempotent] void entire_phone_book(
  [in]    handle_t        h,
  [out]    phone_record_array_t *e_array,
  [out]    error_status_t    *st
  );
```

Next the routine itself, written in C:

```
/* global variables */
dce_db_handle__t db_h; /* handle to phonebook backing store */

/* Other routines are not shown here, including the routine
 * that opened the backing store.
 */
```

```
void
entire_phone_book(
  /* [in] */ handle_t    h, /* For RPC, but not used
                  * here. An ACL check
                  * would use it. */

  /* [out] */ phone_record_array_t *e_array,
  /* [out] */ error_status_t    *st
)
{
  uuid_t       *dbkey;
  phone_data_t    pd;
  unsigned32     i;
  error_status_t   st2;

  *st = error_status_ok;
  /* Lock before starting work, so that the backing
   * store does not change until after all the info
   * has been returned.
   */
  dce_db_lock(db_h, st);
  /* Count the entries so enough storage can be allocated */
  e_array->count = 0;
  dce_db_inq_count(db_h, &e_array->count, st);
  if (*st != error_status_ok) {
    dce_fprintf(stderr, *st); /* or some other treatment */
    dce_db_unlock(db_h, st);
    return;
  }
  if (e_array->count == 0) {  /* No items, nothing to do */
    dce_db_unlock(db_h, st);
    return;
  }
  /* Allocate the space for the output. */
  e_array->entry = rpc_sm_allocate(
        e_array->count*sizeof(e_array->entry[0]),st);
  if (*st != rpc_s_ok) {
    dce_fprintf(stderr, *st); /* or some other treatment */
    return
  }
  dce_db_iter_start(db_h, st);
  i = 0;
  while (TRUE) {
    /* Get the next key. */
    dce_db_iter_next(db_h, &dbkey, st);
    /* break when we've scanned the entire backing store */
    if (*st == db_s_no_more) break;
    /* Get the data associated with the next key. */
    dce_db_fetch_by_uuid(db_h, dbkey, (void *)&pd, st);
    if (*st != error_status_ok) {
      dce_fprintf(stderr, *st);
      /* Don't forget to stop iterating and unlock after
   * an error. */
      dce_db_iter_done(db_h, &st2);
      dce_db_unlock(db_h, &st2);
      return;
    }
    /* Stick the item into the array to be returned
     * when done. */
    e_array->entry[i].name  = strdup(pd.ph.name);
    e_array->entry[i].email = strdup(pd.ph.email);
    e_array->entry[i].phone = strdup(pd.ph.phone);
    e_array->entry[i].office = strdup(pd.ph.office);
    i++;
    /* The use of strdup() above is illustrative, but it
     * is not correct within a server, because the
```

```
 * allocated memory is never freed. Correct code
 * would involve the use of rpc_sm_allocate().
     */
  }
  /* The iteration is finished. */
  dce_db_iter_done(db_h, st);
  dce_db_unlock(db_h, st);
}
```

Part 2. DCE Threads

Chapter 6. Introduction to Multithreaded Programming

DCE Threads is a user-level (nonkernel) threads package based on the pthreads interface specified by POSIX in 1003.4a, Draft 4. This chapter introduces multithreaded programming, which is the division of a program into multiple threads (parts) that execute concurrently. In addition, this chapter describes four software models that improve multithreaded programming performance.

A thread is a single sequential flow of control within a program. It is the active execution of a designated routine, including any nested routine invocations. Within a single thread, there is a single point of execution. Most traditional programs consist of a single thread.

Threads are lightweight processes that share a single address space. Each thread shares all the resources of the originating process, including signal handlers and descriptors. Each thread has its own thread identifier, scheduling policy and priority, **errno** value, thread-specific data bindings, and the required system resources to support a flow of control.

Advantages of Using Threads

With a threads package, a programmer can create multiple threads within a process. Threads execute concurrently and, within a multithreaded process, there are at any time multiple points of execution. Threads execute within a single address space. Multithreaded programming offers the following advantages:

- Performance

  Threads improve the performance (throughput, computational speed, responsiveness, or some combination of these) of a program. Multiple threads are useful in a multiprocessor system where threads run concurrently on separate processors. In addition, multiple threads also improve program performance on single processor systems by permitting the overlap of input and output or other slow operations with computational operations.

  You can think of threads as executing simultaneously, regardless of the number of processors present. You cannot make any assumptions about the start or finish times of threads or the sequence in which they execute, unless explicitly synchronized.
- Shared Resources

  An advantage of using multiple threads over using separate processes is that the former share a single address space, all open files, and other resources.
- Potential Simplicity

  Multiple threads can reduce the complexity of some applications that are inherently suited for threads.

Software Models for Multithreaded Programming

The following subsections describe four software models for which multithreaded programming is especially well suited:

- Boss/worker model
- Work crew model
- Pipelining model
- Combinations of models

Boss/Worker Model

In a boss/worker model of program design, one thread functions as the boss because it assigns tasks to worker threads. Each worker performs a different type of task until it is finished, at which point the worker interrupts the boss to indicate that it is ready to receive another task. Alternatively, the boss polls workers periodically to see whether or not each worker is ready to receive another task.

A variation of the boss/worker model is the work queue model. The boss places tasks in a queue, and workers check the queue and take tasks to perform. An example of the work queue model in an office environment is a secretarial typing pool. The office manager puts documents to be typed in a basket, and typists take documents from the basket to work on.

Work Crew Model

In the work crew model, multiple threads work together on a single task. The task is divided into pieces that are performed in parallel, and each thread performs one piece. An example of a work crew is a group of people cleaning a house. Each person cleans certain rooms or performs certain types of work (washing floors, polishing furniture, and so forth), and each works independently. Figure 6 shows a task performed by three threads in a work crew model.

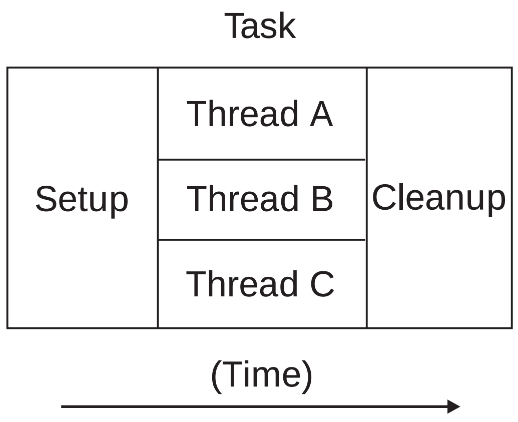


Figure 6. Work Crew Model

Pipelining Model

In the pipelining model, a task is divided into steps. The steps must be performed in sequence to produce a single instance of the desired output, and the work done in each step (except for the first and last) is based on the preceding step and is a prerequisite for the work in the next step. However, the program is designed to produce multiple instances of the desired output, and the steps are designed to operate in a parallel time frame so that each step is kept busy.

An example of the pipelining model is an automobile assembly line. Each step or stage in the assembly line is continually busy receiving the product of the previous stage's work, performing its assigned work, and passing the product along to the next stage. A car needs a body before it can be painted, but at any one time numerous cars are receiving bodies, and then numerous cars are being painted.

In a multithreaded program using the pipelining model, each thread represents a step in the task. Figure 7 shows a task performed by three threads in a pipelining model.

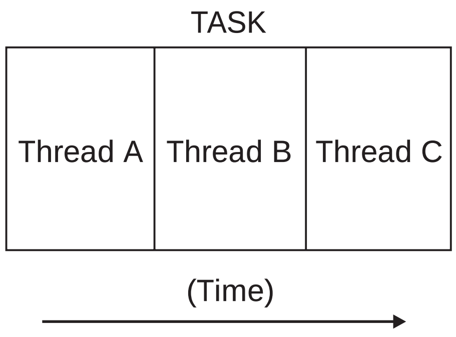


Figure 7. Pipelining Model

Combinations of Models

You may find it appropriate to combine the software models in a single program if your task is complex. For example, a program could be designed using the pipelining model, but one or more steps could be handled by a work crew. In addition, tasks could be assigned to a work crew by taking a task from a work queue and deciding (based on the task characteristics) which threads are needed for the work crew.

Potential Disadvantages of Multithreaded Programming

When you design and code a multithreaded program, consider the following problems and accommodate or eliminate each problem as appropriate:

- Potential Complexity

  The level of expertise required for designing, coding, and maintaining multithreaded programs may be higher than for most single-threaded programs because multithreaded programs may need shared access to resources, mutexes, and condition variables. Weigh the potential benefits against the complexity and its associated risks.
- Nonreentrant Software

  If a thread calls a routine or library that is not reentrant, use the global locking mechanism to prevent the nonreentrant routines from modifying a variable that another thread modifies. “Chapter 8. Programming with Threads” on page 121 discusses nonreentrant software in more detail.

  Note: A multithreaded program must be reentrant; that is, it must allow multiple threads to execute at the same time. Therefore, be sure that your compiler generates reentrant code before you do any design or coding work for multithreading. (Many C, Ada, Pascal, and BLISS compilers generate reentrant code by default.)

  If your program is nonreentrant, any thread synchronization techniques that you use are not guaranteed to be effective.
- Priority Inversion

Priority inversion prevents high-priority threads from executing when interdependencies exist among three or more threads. “Chapter 8. Programming with Threads” on page 121 discusses priority inversion in more detail.

- Race Conditions

  A type of programming error called a race condition causes unpredictable and erroneous program behavior. “Chapter 8. Programming with Threads” on page 121 discusses race conditions in more detail.

- Deadlocks

  A type of programming error called a deadlock causes two or more threads to be blocked from executing. “Chapter 8. Programming with Threads” on page 121 discusses deadlocks in more detail.

- Blocking Calls

  Certain system or library calls may cause an entire process to block while waiting for the call to complete, thus causing all other threads to stop executing. “Chapter 8. Programming with Threads” on page 121 discusses blocking in more detail.

Chapter 7. Thread Concepts and Operations

This chapter discusses concepts and techniques related to DCE Threads. The following topics are covered:

- Thread operations
- New primitives
- Attributes objects
- Synchronization objects
- One-time initialization code
- Thread-specific data
- Thread cancellation
- Thread scheduling

For detailed information on the multithreading routines referred to in this chapter, see the reference page for that routine in the *OSF DCE Application Development Reference*.

Thread Operations

A thread changes states as it runs, waits to synchronize, or is ready to be run. A thread is in one of the following states:

- Waiting

  The thread is not eligible to execute because it is synchronizing with another thread or with an external event.
- Ready

  The thread is eligible to be executed by a processor.
- Running

  The thread is currently being executed by a processor.
- Terminated

  The thread has completed all of its work.

Figure 8 shows the transitions between states for a typical thread implementation.

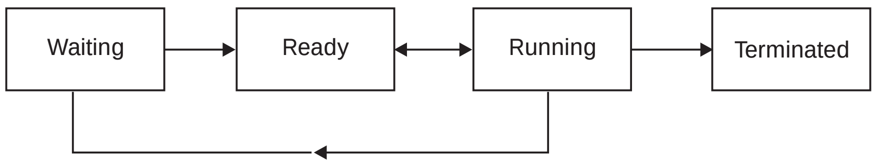


Figure 8. Thread State Transitions

The operations that you can perform include starting, waiting for, terminating, and deleting threads.

Starting a Thread

To start a thread, create it using the **pthread_create()** routine. This routine creates the thread, assigns specified or default attributes, and starts execution of the function you specified as the thread’s start routine. A unique identifier (handle) for that thread is returned from the **pthread_create()** routine.

Terminating a Thread

A thread exists until it terminates and the **pthread_detach()** routine is called for the thread. The **pthread_detach()** routine can be called for a thread before or after it terminates. If the thread terminates before **pthread_detach()** is called for it, then the thread continues to exist and can be synchronized (joined) until it is detached. Thus, the object (thread) can be detached by any thread that has access to a handle to the object.

Note that **pthread_detach()** must be called to release the memory allocated for the thread objects so that this storage does not build up and cause the process to run out of memory. For example, after a thread returns from a call to join, it detaches the joined-to thread if no other threads join with it. Similarly, if a thread has no other threads joining with it, it detaches itself so that its thread object is deallocated as soon as it terminates.

A thread terminates for any of the following reasons:

- The thread returns from its start routine; this is the usual case.
- The thread calls the **pthread_exit()** routine.

  The **pthread_exit()** routine terminates the calling thread and returns a status value, indicating the thread’s exit status to any potential joiners.
- The thread is canceled by a call to the **pthread_cancel()** routine.

  The **pthread_cancel()** routine requests termination of a specified thread if cancellation is permitted. (See “Thread Cancellation” on page 117 for more information on canceling threads and controlling whether or not cancellation is permitted.)
- An error occurs in the thread.

  Examples of errors that cause thread termination are programming errors, segmentation faults, or unhandled exceptions.

Waiting for a Thread to Terminate

A thread waits for the termination of another thread by calling the **pthread_join()** routine. Execution in the current thread is suspended until the specified thread terminates. If multiple threads call this routine and specify the same thread, all threads resume execution when the specified thread terminates.

If you specify the current thread with the **pthread_join()** routine, a deadlock results.

Do not confuse **pthread_join()** with other routines that cause waits and that are related to the use of a particular multithreading feature. For example, use **pthread_cond_wait()** or **pthread_cond_timedwait()** to wait for a condition variable to be signaled or broadcast.

Deleting a Thread

A thread is automatically deleted after it terminates; that is, no explicit deletion operation is required. Use **pthread_detach()** to free the storage of a terminated thread. Use **pthread_cancel()** to request that a running thread terminate itself.

If the thread has not yet terminated, the **pthread_detach()** routine marks the thread for deletion, and its storage is reclaimed immediately when the thread terminates. A thread cannot be joined or canceled after the **pthread_detach()** routine is called for the thread, even if the thread has not yet terminated.

If a thread that is not detached terminates, its storage remains so that other threads can join with it. Storage is reclaimed when the thread is eventually detached.

New Primitives

Routines implemented by DCE Threads that are not specified by Draft 4 of the POSIX 1003.4a standard are indicated by an **_np** suffix to the name. These routines have not been incorporated into the POSIX standard, and as such are extensions to that document. The routines are fully portable.

Attributes Objects

An attributes object is used to describe the behavior of threads, mutexes, and condition variables. This description consists of the individual attribute values that are used to create an attributes object. Whether an attribute is valid depends on whether it describes threads, mutexes, or condition variables.

When you create an object, you can accept the default attributes for that object, or you can specify an attributes object that contains individual attributes that you have set. For a thread, you can also change one or more attributes after thread execution starts; for example, calling the **pthread_setprio()** routine to change the priority that you specified with the **pthread_attr_setprio()** routine.

The following subsections describe how to create and delete attributes objects and describe the individual attributes that you can specify for different objects.

Creating an Attributes Object

To create an attributes object, use one of the following routines, depending on the type of object to which the attributes apply:

- The **pthread_attr_create()** routine for thread attributes objects
- The **pthread_condattr_create()** routine for condition variable attributes objects
- The **pthread_mutexattr_create()** routine for mutex attributes objects

These routines create an attributes object containing default values for the individual attributes. To modify any attribute values in an attributes object, use one of the set routines described in the following subsections.

Creating an attributes object or changing the values in an attributes object does not affect the attributes of objects previously created.

Deleting an Attributes Object

To delete an attributes object, use one of the following routines:

- The **pthread_attr_delete()** routine for thread attributes objects
- The **pthread_condattr_delete()** routine for condition variable attributes objects
- The **pthread_mutexattr_delete()** routine for mutex attributes objects

Deleting an attributes object does not affect the attributes of objects previously created.

Thread Attributes

A thread attributes object allows you to specify values for thread attributes other than the defaults when you create a thread with the **pthread_create()** routine. To use a thread attributes object, perform the following steps:

1. Create a thread attributes object by calling the routine **pthread_attr_create()**.
2. Call the routines discussed in the following subsections to set the individual attributes of the thread attributes object.
3. Create a new thread by calling the **pthread_create()** routine and specifying the identifier of the thread attributes object.

You have control over the following attributes of a new thread:

- Scheduling policy attribute
- Scheduling priority attribute
- Inherit scheduling attribute
- Stacksize attribute

Scheduling Policy Attribute

The scheduling policy attribute describes the overall scheduling policy of the threads in your application. A thread has one of the following scheduling policies:

- **SCHED_FIFO** (First In, First Out)

  The highest-priority thread runs until it blocks. If there is more than one thread with the same priority, and that priority is the highest among other threads, the first thread to begin running continues until it blocks.
- **SCHED_RR** (Round Robin)

  The highest-priority thread runs until it blocks; however, threads of equal priority, if that priority is the highest among other threads, are timesliced. (Timeslicing is a mechanism that ensures that every thread is allowed time to execute by preempting running threads at fixed intervals.)
- **SCHED_OTHER, SCHED_FG_NP** (Default)

  All threads are timesliced. **SCHED_OTHER** and **SCHED_FG_NP** do the same thing; however, **SCHED_FG_NP** is simply more precise terminology. The **FG** stands for *foreground* and the **NP** for new primitive. All threads running under the **SCHED_OTHER** and **SCHED_FG_NP** policy, regardless of priority, receive some scheduling. Therefore, no thread is completely denied execution time. However, **SCHED_OTHER** and **SCHED_FG_NP** threads can be denied execution time by **SCHED_FIFO** or **SCHED_RR** threads.
- **SCHED_BG_NP** (Background)

  Like **SCHED_OTHER** and **SCHED_FG_NP**, **SCHED_BG_NP** ensures that all threads, regardless of priority, receive some scheduling. However, **SCHED_BG_NP** can be denied execution by the **SCHED_FIFO** or **SCHED_RR** policies. The **BG** stands for *background*.

The following two methods are used to set the scheduling policy attribute:

- Set the scheduling policy attribute in the attributes object, which establishes the scheduling policy of a new thread when it is created. To do this, call the **pthread_attr_setsched()** routine.

- Change the scheduling policy of an existing thread (and, at the same time, the scheduling priority) by calling the **pthread_setscheduler()** routine.

“Thread Scheduling” on page 118 describes and shows the effect of scheduling policy on thread scheduling.

Scheduling Priority Attribute

The scheduling priority attribute specifies the execution of a thread. This attribute is expressed relative to other threads on a continuum of minimum to maximum for each scheduling policy. A thread’s priority falls within one of the following ranges, which are implementation defined:

- **PRI_FIFO_MIN** to **PRI_FIFO_MAX**
- **PRI_RR_MIN** to **PRI_RR_MAX**
- **PRI_OTHER_MIN** to **PRI_OTHER_MAX**
- **PRI_FG_MIN_NP** to **PRI_FG_MAX_NP**
- **PRI_BG_MIN_NP** to **PRI_BG_MAX_NP**

The following two methods are used to set the scheduling priority attribute:

- Set the scheduling priority attribute in the attributes object, which establishes the execution priority of a new thread when it is created. To do this, call the **pthread_attr_setprio()** routine.
- Change the scheduling priority attribute of an existing thread by calling the **pthread_setprio()** routine. (Call the **pthread_setscheduler()** routine to change both the scheduling priority and scheduling policy of an existing thread.)

Inherit Scheduling Attribute

The inherit scheduling attribute specifies whether a newly created thread inherits the scheduling attributes (scheduling priority and policy) of the creating thread (the default), or uses the scheduling attributes stored in the attributes object. Set this attribute by calling the routine **pthread_attr_setinheritsched()**.

Stacksize Attribute

The stacksize attribute is the minimum size (in bytes) of the memory required for a thread’s stack. The default value is machine dependent. Set this attribute by calling the **pthread_attr_setstacksize()** routine.

Mutex Attributes

A mutex attributes object allows you to specify values for mutex attributes other than the defaults when you create a mutex with the routine **pthread_mutex_init()**.

The mutex type attribute specifies whether a mutex is fast, recursive, or nonrecursive. Set the mutex type attribute by calling the routine **pthread_mutexattr_setkind_np()**. (Any routine with the **_np** suffix is a new primitive; see “New Primitives” on page 109.) If you do not use a mutex attributes object to select a mutex type, calling the **pthread_mutex_init()** routine creates a fast mutex by default.

Condition Variable Attributes

Currently, attributes affecting condition variables are not defined. You cannot change any attributes in the condition variable attributes object.

"Condition Variables" on page 113 describes the purpose and uses of condition variables.

Synchronization Objects

In a multithreaded program, you must use synchronization objects whenever there is a possibility of corruption of shared data or conflicting scheduling of threads that have mutual scheduling dependencies. The following subsections discuss two kinds of synchronization objects: mutexes and condition variables.

Mutexes

A mutex (*mut*ual *ex*clusion) is an object that multiple threads use to ensure the integrity of a shared resource that they access, most commonly shared data. A mutex has two states: locked and unlocked. For each piece of shared data, all threads accessing that data must use the same mutex; each thread locks the mutex before it accesses the shared data and unlocks the mutex when it is finished accessing that data. If the mutex is locked by another thread, the thread requesting the lock is blocked when it tries to lock the mutex if you call **pthread_mutex_lock()** (see Figure 9). The blocked thread continues and is not blocked if you call **pthread_mutex_trylock()**.

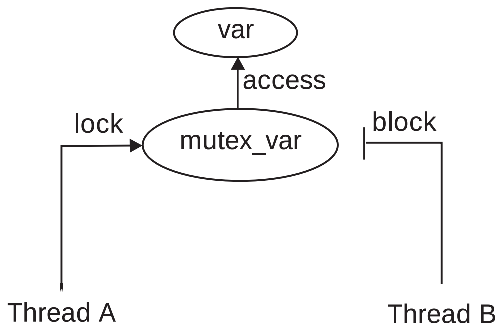


Figure 9. Only One Thread Can Lock a Mutex

Each mutex must be initialized. (To initialize mutexes as part of the program's one-time initialization code, see "One-Time Initialization Routines" on page 116.) To initialize a mutex, use the **pthread_mutex_init()** routine. This routine allows you to specify an attributes object, which allows you to specify the mutex type. The following are types of mutexes:

- A fast mutex (the default) is locked only once by a thread. If the thread tries to lock the mutex again without first unlocking it, the thread waits for itself to release the first lock and deadlocks on itself.

  This type of mutex is called fast because it can be locked and unlocked more rapidly than a recursive mutex. It is the most efficient form of mutex.
- A recursive mutex can be locked more than once by a given thread without causing a deadlock. The thread must call the **pthread_mutex_unlock()** routine

the same number of times that it called the **pthread_mutex_lock()** routine before another thread can lock the mutex. Recursive mutexes have the notion of a mutex owner. When a thread successfully locks a recursive mutex, it owns that mutex and the lock count is set to 1. Any other thread attempting to lock the mutex blocks until the mutex becomes unlocked. If the owner of the mutex attempts to lock the mutex again, the lock count is incremented, and the thread continues running. When an owner unlocks a recursive mutex, the lock count is decremented. The mutex remains locked and owned until the count reaches 0 (zero). It is an error for any thread other than the owner to attempt to unlock the mutex.

  A recursive mutex is useful if a thread needs exclusive access to a piece of data, and it needs to call another routine (or itself) that needs exclusive access to the data. A recursive mutex allows nested attempts to lock the mutex to succeed rather than deadlock.

  This type of mutex requires more careful programming. Never use a recursive mutex with condition variables because the implicit unlock performed for a **pthread_cond_wait()** or **pthread_cond_timedwait()** may not actually release the mutex. In that case, no other thread can satisfy the condition of the predicate.

- A nonrecursive mutex is locked only once by a thread, like a fast mutex. If the thread tries to lock the mutex again without first unlocking it, the thread receives an error. Thus, nonrecursive mutexes are more informative than fast mutexes because fast mutexes block in such a case, leaving it up to you to determine why the thread no longer executes. Also, if someone other than the owner tries to unlock a nonrecursive mutex, an error is returned.

To lock a mutex, use one of the following routines, depending on what you want to happen if the mutex is locked:

- The **pthread_mutex_lock()** routine

  If the mutex is locked, the thread waits for the mutex to become available.

- The **pthread_mutex_trylock()** routine

  If the mutex is locked, the thread continues without waiting for the mutex to become available. The thread immediately checks the return status to see if the lock was successful, and then takes whatever action is appropriate if it was not.

When a thread is finished accessing a piece of shared data, it unlocks the associated mutex by calling the **pthread_mutex_unlock()** routine.

If another thread is waiting on the mutex, its execution is unblocked. If more than one thread is waiting on the mutex, the scheduling policy and the thread scheduling priority determine which thread acquires the mutex.

You can delete a mutex and reclaim its storage by calling the **pthread_mutex_destroy()** routine. Use this routine only after the mutex is no longer needed by any thread. Mutexes are automatically deleted when the program terminates.

Condition Variables

A condition variable allows a thread to block its own execution until some shared data reaches a particular state. Cooperating threads check the shared data and wait on the condition variable. For example, one thread in a program produces work-to-do packets and another thread consumes these packets (does the work). If the work queue is empty when the consumer thread checks it, that thread waits on a work-to-do condition variable. When the producer thread puts a packet on the queue, it signals the work-to-do condition variable.

A condition variable is used to wait for a shared resource to assume some specific state (a predicate). A mutex, on the other hand, is used to reserve some shared resource while the resource is being manipulated. For example, a thread A may need to wait for a thread B to finish a task X before thread A proceeds to execute a task Y. Thread B can tell thread A that it has finished task X by using a variable they both have access to, a condition variable . When thread A is ready to execute task Y, it looks at the condition variable to see if thread B is finished (see Figure 10).

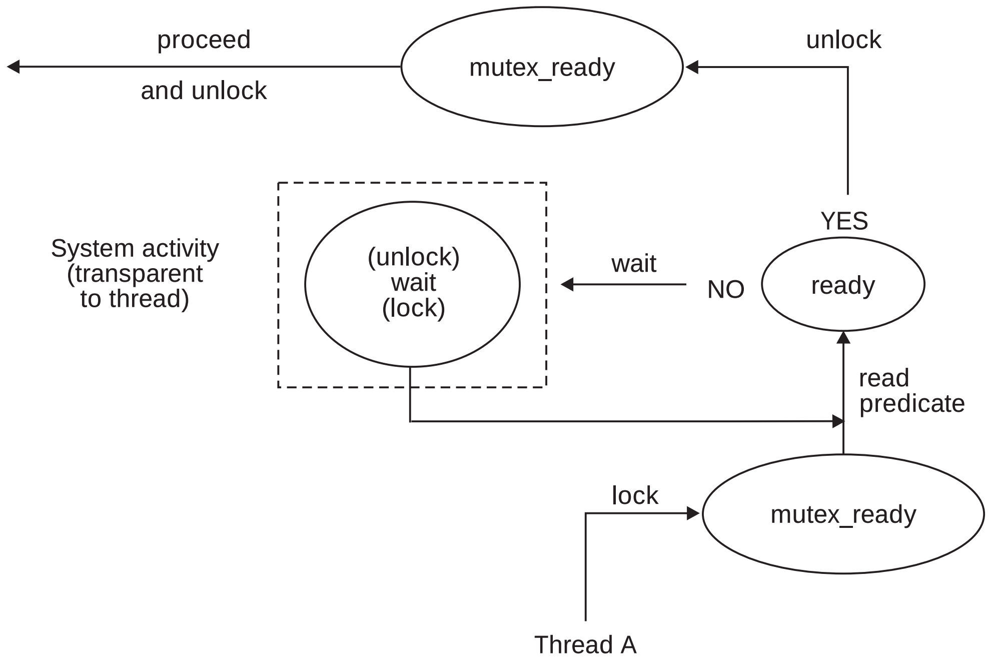


Figure 10. Thread A Waits on Condition Ready, Then Wakes Up and Proceeds

First, thread A locks the mutex named **mutex_ready** that is associated with the condition variable. Then it reads the predicate associated with the condition variable named **ready**. If the predicate indicates that thread B has finished task X, then thread A can unlock the mutex and proceed with task Y. If the condition variable predicate indicated that thread B has not yet finished task X; however, then thread A waits for the condition variable to change. Thread A calls the **wait** primitive. Waiting on the condition variable automatically unlocks the mutex, allowing thread B to lock the mutex when it has finished task X. (see Figure 11 on page 115).

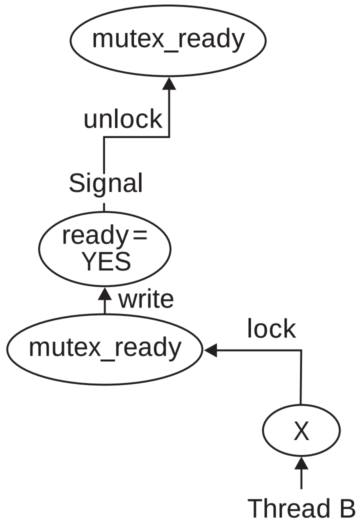


Figure 11. Thread B Signals Condition Ready

Thread B updates the predicate named **ready** associated with the condition variable to the state thread A is waiting for. It also executes a signal on the condition variable while holding the mutex **mutex_ready**.

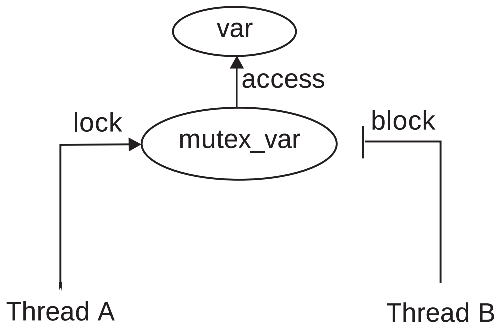


Figure 12. Thread A Wakes Up and Proceeds

Thread A wakes up, verifies that the condition variable is in the correct state, and proceeds to execute task Y (see Figure 10 on page 114).

Note that, although the condition variable is used for explicit communications among threads, the communications are anonymous. Thread B does not necessarily know that thread A is waiting on the condition variable that thread B signals. And thread A does not know that it was thread B that woke it up from its wait on the condition variable.

Use the **pthread_cond_init()** routine to create a condition variable. To create condition variables as part of the program's one-time initialization code, see "One-Time Initialization Routines" on page 116.

Use the **pthread_cond_wait()** routine to cause a thread to wait until the condition is signaled or broadcast. This routine specifies a condition variable and a mutex that you have locked. (If you have not locked the mutex, the results of **pthread_cond_wait()** are unpredictable.) This routine unlocks the mutex and causes the calling thread to wait on the condition variable until another thread calls one of the following routines:

- The **pthread_cond_signal()** routine to wake one thread that is waiting on the condition variable
- The **pthread_cond_broadcast()** routine to wake all threads that are waiting on a condition variable

If you want to limit the time that a thread waits for a condition to be signaled or broadcast, use the **pthread_cond_timedwait()** routine. This routine specifies the condition variable, mutex, and absolute time at which the wait should expire if the condition variable is not signaled or broadcast.

You can delete a condition variable and reclaim its storage by calling the **pthread_cond_destroy()** routine. Use this routine only after the condition variable is no longer needed by any thread. Condition variables are automatically deleted when the program terminates.

Other Synchronization Methods

There is another synchronization method that is not anonymous: the **join** primitive. This allows a thread to wait for another specific thread to complete its execution. When the second thread is finished, the first thread unblocks and continues its execution. Unlike mutexes and condition variables, the **join** primitive is not associated with any particular shared data.

One-Time Initialization Routines

You probably have one or more routines that must be executed *before* any thread executes code in your application, but must be executed only once regardless of the sequence in which threads start executing. For example, you may want to create mutexes and condition variables (each of which must be created only once) in an initialization routine. Multiple threads can call the **pthread_once()** routine, or the **pthread_once()** routine can be called multiple times in the same thread, resulting in only one call to the specified routine.

Use the **pthread_once()** routine to ensure that your application initialization routine is executed only a single time, that is, by the first thread that tries to initialize the application. This routine is the only way to guarantee that one-time initialization is performed in a multithreaded environment on a given platform. The **pthread_once()** routine is of particular use for runtime libraries, which are often called for the first time after multiple threads are created.

Refer to the **thr_intro(3thr)** reference page for a list of the DCE Threads routines which, when called, implicitly perform any necessary initialization of the threads package. Any application that uses DCE Threads must call one of these routines before calling any other threads routines.

Thread-Specific Data

The thread-specific data interfaces allow each thread to associate an arbitrary value with a shared key value created by the program.

Thread-specific data is like a global variable in which each thread can keep its own value, but is accessible to the thread anywhere in the program.

Use the following routines to create and access thread-specific data:

- The **pthread_keycreate()** routine to create a unique key value
- The **pthread_setspecific()** routine to associate data with a key
- The **pthread_getspecific()** routine to obtain the data associated with a key

The **pthread_keycreate()** routine generates a unique key value that is shared by all threads in the process. This key is the identifier of a piece of thread-specific data. Each thread uses the same key value to assign or retrieve a thread-specific value. This keeps your data separate from other thread-specific data. One call to the **pthread_keycreate()** routine creates a cell in all threads. Call this routine to specify a routine to be called to destroy the context value associated with this key when the thread terminates.

The **pthread_setspecific()** routine associates the address of some data with a specific key. Multiple threads associate different data (by specifying different addresses) with the same key. For example, each thread points to a different block of dynamically allocated memory that it has reserved.

The **pthread_getspecific()** routine obtains the address of the thread-specific data value associated with a specified key. Use this routine to locate the data associated with the current thread's context.

Thread Cancellation

Canceling is a mechanism by which one thread terminates another thread (or itself). When you request that a thread be canceled, you are requesting that it terminate as soon as possible. However, the target thread can control how quickly it terminates by controlling its general cancelability and its asynchronous cancelability.

The following is a list of the pthread calls that are cancellation points:
- The **pthread_setasynccancel()** routine
- The **pthread_testcancel()** routine
- The **pthread_delay_np()** routine
- The **pthread_join()** routine
- The **pthread_cond_wait()** routine
- The **pthread_cond_timedwait()** routine

General cancelability is enabled by default. A thread is canceled only at specific places in the program; for example, when a call to the **pthread_cond_wait()** routine is made. If general cancelability is enabled, request the delivery of any pending cancel request by using the **pthread_testcancel()** routine. This routine allows you to permit cancellation to occur at places where it may not otherwise be permitted under general cancelability, and it is especially useful within very long loops to ensure that cancel requests are noticed within a reasonable time.

If you disable general cancelability, the thread cannot be terminated by any cancel request. Disabling general cancelability means that a thread could wait indefinitely if it does not come to a normal conclusion. Therefore, be careful about disabling general cancelability.

Asynchronous cancelability, when it is enabled, allows cancels to be delivered to the enabling thread at any time, not only at those times that are permitted when just general cancelability is enabled. Thus, use asynchronous cancellation primarily during long processes that do not have specific places for cancel requests.

Asynchronous cancelability is disabled by default. Disable asynchronous cancelability when calling threads routines or any other runtime library routines that are not explicitly documented as cancel-safe.

Note: If general cancelability is disabled, the thread cannot be canceled, regardless of whether asynchronous cancelability is enabled or disabled. The setting of asynchronous cancelability is relevant only when general cancelability is enabled.

Use the following routines to control the canceling of threads:
- The **pthread_setcancel()** routine to enable and disable general cancelability
- The **pthread_testcancel()** routine to request delivery of a pending cancel to the current thread
- The **pthread_setasynccancel()** routine to enable and disable asynchronous cancelability
- The **pthread_cancel()** routine to request that a thread be canceled

Thread Scheduling

Threads are scheduled according to their scheduling priority and how the scheduling policy treats those priorities. To understand the discussion in this section, you must understand the concepts in the following sections of this chapter:
- “Scheduling Policy Attribute” on page 110 discusses scheduling policies, including the way in which each policy handles thread scheduling priority.
- “Scheduling Priority Attribute” on page 111 discusses thread scheduling priorities.
- “Inherit Scheduling Attribute” on page 111 discusses inheritance of scheduling attributes by created threads.

To specify the minimum or maximum priority, use the appropriate symbol; for example, **PRI_OTHER_MIN** or **PRI_OTHER_MAX**. To specify a value between the minimum and maximum priority, use an appropriate arithmetic expression.

For example, to specify a priority midway between the minimum and maximum for the default scheduling policy, specify the following concept using your programming language’s syntax:

```
pri_other_mid = (PRI_OTHER_MIN + PRI_OTHER_MAX)/2
```

If your expression results in a value outside the range of minimum to maximum, an error results when you use it. Priority values are integers.

To show results of the different scheduling policies, consider the following example: a program has four threads, called threads A, B, C, and D. For each scheduling policy, three scheduling priorities have been defined: minimum, middle, and maximum. The threads have the priorities shown in Table 5.

Table 5. Sample Thread Properties

Thread	**Priority**
A	Minimum
B	Middle
C	Middle

Table 5. Sample Thread Properties (continued)

D	Maximum

Figure 13 through Figure 15 show execution flows, depending on whether the threads use the **SCHED_FIFO**, **SCHED_RR**, or **SCHED_OTHER** (default) scheduling policy. Assume that all waiting threads are ready to execute when the current thread waits or terminates and that no higher-priority thread is awakened while a thread is executing (during the flow shown in each figure).

Figure 13 shows a flow with **SCHED_FIFO** (First In, First Out) scheduling.

$$\begin{tikzcd}
D \arrow[r] & B \arrow[r] & C \arrow[r] & A \arrow[r] & {}
\end{tikzcd}$$

Figure 13. Flow with SCHED_FIFO Scheduling

Thread D executes until it waits or terminates, then Thread B starts because it has been waiting longer than Thread C and it executes until it waits or terminates, then Thread C executes until it waits or terminates, then Thread A executes.

Figure 14 shows a flow with **SCHED_RR** (Round Robin) scheduling.

$$\begin{tikzcd}
D \arrow[r] & B \arrow[r] & C \arrow[r] & B \arrow[r] & C \arrow[r] & A \arrow[r] & {}
\end{tikzcd}$$

Figure 14. Flow with SCHED_RR Scheduling

All four threads are timesliced. Threads with higher priority are generally scheduled when more than one thread is ready to run; however, to ensure fairness, all threads are given some time. The effective priority of threads may be modified over time by the scheduler, depending on the use of processor resources.

Thread D executes until it waits or terminates, then threads B and C are timesliced because they both have middle priority, then thread A executes.

Figure 15 shows a flow with **SCHED_OTHER** (default) scheduling.

$$\begin{tikzcd}
D \arrow[r] & B \arrow[r] & C \arrow[r] & A \arrow[r] & B \arrow[r] & C \arrow[r] & \cdots
\end{tikzcd}$$

Figure 15. Flow with SCHED_OTHER Scheduling

Thread D executes until it waits or terminates; then threads B, C, and A are timesliced, even though thread A has a lower priority than the other two. Thread A receives less execution time than thread B or C if either is ready to execute as often as thread A is. However, the default scheduling policy protects thread A against being blocked from executing indefinitely.

Because low-priority threads eventually run, the default scheduling policy protects against the problem of priority inversion discussed in “Chapter 8. Programming with Threads” on page 121.

Chapter 8. Programming with Threads

This chapter discusses issues you face when writing a multithreaded program and how to deal with those issues.

The topics discussed in this chapter are as follows:
- Calling UNIX services
- Using signals
- Nonthreaded libraries
- Avoiding nonreentrant software
- Avoiding priority inversion
- Using synchronization objects
- Signaling a condition variable

Calling UNIX Services

On a UNIX system that does not have kernel support for threads, making system and library calls from within a multithreaded program raises the following issues:
- System calls may not be thread-reentrant.
- If a system call blocks, it blocks the entire process instead of blocking the calling thread only.

Jacket Routines

To resolve the previous two issues, DCE Threads provides jacket routines for a number of UNIX system calls. Threads call the jacket routine instead of the UNIX system service; this allows DCE Threads to take action on behalf of the thread before or after calling the system service. For example, the jacket routines ensure that only one thread calls any particular service at a time to avoid problems with system calls that are not thread-reentrant.

Jacket routines are provided for UNIX input and output system calls (documented in any UNIX programmer’s manual) and the **fork()** and **sigaction()** system calls. Jackets are not provided for any other UNIX system calls or for any of the C runtime library services. See **/usr/include/dce/cma_ux.h** for the full list of jacket routines.

Input and Output Jacket Routines

Jacket routines are provided for routines that perform input and output operations. Examples of these operations are as follows:
1. Open or create files, pipe symbols, and sockets
2. Send and receive messages on sockets
3. Read and write files and pipe symbols

Jacket routines are provided for Input/Output services so that DCE Threads can determine when to issue or block the service call based on the results of the **select()** system call. For these UNIX services, DCE Threads can determine whether issuing the system call causes the process to block. If the system call causes the process to block, DCE Threads blocks only the calling thread and schedules another thread to run in its place.

Periodically, DCE Threads checks whether the original calling thread can issue its operation without blocking the process. When the thread runs without blocking the process, that thread is placed back into the queue of ready threads and, at its turn, the thread resumes execution and issues the system call. Therefore, the jacket routines provide thread-synchronous I/O operations where otherwise the system calls block the entire process.

The fork() Jacket Routine

Jackets are provided for the **fork()** system call. A specific thread environment must exist in the forked process when it resumes (begins) execution. These jacket routines allow code to be executed in the context of the new process before the user code resumes execution in it.

The atfork() Routine

The **atfork()** routine allows an application or library to ensure predicted behavior when the **fork()** routine is used in a multithreaded environment. Using the **fork()** routine from a threaded application or from an application that uses threaded libraries can result in unpredictable behavior. For example, one thread has a mutex locked, and the state covered by that mutex is inconsistent while another thread calls the **fork()** routine. In the child process, the mutex will be in the locked state, and it cannot be unlocked because only the forking thread exists in the child process. Having the child reinitialize the mutex is unsatisfactory because this approach does not resolve the question of how to correct the inconsistent state in the child.

The **atfork()** routine provides a way for threaded applications or libraries to protect themselves when a **fork()** occurs. The **atfork()** routine allows you to set up routines that will run at the following times:

- Prior to the **fork()** in the parent process
- After the **fork()** in the child process
- After the **fork()** in the parent process

Within these routines, you can ensure that all mutexes are locked prior to the **fork()** and that they are unlocked after the **fork()**, thereby protecting any data or resources associated with the mutexes. You can register any number of sets of **atfork()** routines; that is, any number of libraries or user programs can set up **atfork()** routines and they will all execute at **fork()** time.

Note: Using the **atfork()** routine can potentially cause a deadlock if two applications or libraries call into one another using calls that require locking. Specifically, when these component's routines use the **atfork()** routine to run prior to the fork in the parent process, a deadlock may occur when these routines are executing.

Using the Jacketed System Calls

You do not have to rename your system calls to take advantage of the jacket routines. Macros put the jacket routines into place when you compile your program; these macros rename the jacketed system calls to the name of the DCE Threads jacket routine. Thus, a reference to the DCE Threads jacket routine is compiled into your code instead of a reference to the system call. When the code is executed, it calls the jacket routine, which then calls the system on your code's behalf.

If you do not wish to use any of the jacket routines, you can add the following line to your program before any of the thread header files:

```
#define _CMA_NOWRAPPERS_
```

By adding this definition, you prevent the jacket routines from being substituted for the real routines.

If you wish to use most of the jackets but do not wish to use a specific jacket, you can undefine a specific jacket by adding the following directive after the thread header files:

```
#undef routine_name
```

For example, to not use the fork jacket, you can add the following:

```
#undef fork
```

Blocking System Calls

DCE Threads provides jacket routines that make certain system calls thread-synchronous. If calling one of these jacketed system calls would normally block the process, the jacket routine ensures that only the calling thread is blocked and that the process remains available to execute other threads. Examples of jacketed system calls include **read()**, **write()**, **open()**, **socket()**, **send()**, and **recv()**.

If a thread makes a call to any of the other nonjacketed blocking system calls (or if it calls one of the jacketed system calls without going through the jacket), then when the system call blocks the thread, it blocks the whole process, preventing any other threads in the process from executing. Examples of nonjacketed system calls include **wait()**, **sigpause()**, **msgsnd()**, **msgrcv()**, and **semop()**.

Some care must be used when calling nonjacketed blocking system calls from a multithreaded program. Other threads in the program may not be able to tolerate not running for an extended period of time while the process blocks for the system call. If your program must make use of such system calls, the calling thread should specify a nonblocking or polling option to the system call. If the call is not successful, then the calling thread should retry; however, to prevent the retry code from becoming a hot loop, a yield or delay function call should be inserted into the path. This gives other threads in the program a chance to run between poll attempts.

Calling fork() in a Multithreaded Environment

The **fork()** system call creates an exact duplicate of the address space from which it is called, resulting in two address spaces executing the same code. Problems can occur if the forking address space has multiple threads executing at the time of the **fork()**. When multithreading is a result of library invocation, threads are not necessarily aware of each other's presence, purpose, actions, and so on. Suppose that one of the other threads (any thread other than the one doing the **fork()**) has the job of deducting money from your checking account. Clearly, you do not want this to happen twice as a result of some other thread's decision to call **fork()**.

Because of these types of problems, which in general are problems of threads modifying persistent state, POSIX defined the behavior of **fork()** in the presence of threads to propagate only the forking thread. This solves the problem of improper changes being made to persistent state. However, it causes other problems, as discussed in the next paragraph.

In the POSIX model, only the forking thread is propagated. All the other threads are eliminated without any form of notice; no cancels are sent and no handlers are run. However, all the other portions of the address space are cloned, including all the mutex state. If the other thread has a mutex locked, the mutex will be locked in the child process, but the lock owner will not exist to unlock it. Therefore, the resource protected by the lock will be permanently unavailable.

The fact that there may be mutexes outstanding only becomes a problem if your code attempts to lock a mutex that could be locked by another thread at the time of the **fork()**. This means that you cannot call outside of your own code between the call to **fork()** and the call to **exec()**. Note that a call to **malloc()**, for example, is a call outside of the currently executing application program and may have a mutex outstanding. The following code obeys these guidelines and is therefore safe:

```
fork ();
a = 1+2; /* some inline processing */
exec();
```

Similarly, if your code calls some of your own code that does not make any calls outside of your code and does not lock any mutexes that could possibly be locked in another thread, then your code is safe.

One solution to the problem of calling **fork()** in a multithreaded environment exists. (Note that this method will not work for server application code or any other application code that is invoked by a callback from a library.) Before an application performs a **fork()** followed by something other than **exec()**, it must cancel all of the other threads. After it joins the canceled threads, it can safely **fork()** because it is the only thread in existence. This means that libraries that create threads must establish cancel handlers that propagate the cancel to the created threads and join them. The application should save enough state so that the threads can be recreated and restarted after the **fork()** processing completes.

Using Signals

The following subsections cover three topics: types of signals, DCE Threads signal handling, and alternatives to using signals.

Types of Signals

Signals are delivered as a result of some event. UNIX signals are grouped into the following four categories of pairs that are orthogonal to each other:

- Terminating and synchronous
- Terminating and asynchronous
- Nonterminating and synchronous
- Nonterminating and asynchronous

The action that DCE Threads takes when a particular signal is delivered depends on the characteristics of that signal.

Terminating Signals

Terminating signals result in the termination of the process by default. Whether a particular signal is terminating or not is independent of whether it is synchronously or asynchronously delivered.

Nonterminating Signals

Nonterminating signals do not result in the termination of the process by default.

Nonterminating signals represent events that can be either internal or external to the process. The process may require notification or ignore these events. When a nonterminating asynchronous signal is delivered to the process, DCE Threads awakens any threads that are waiting for the signal. This is the only action that DCE Threads takes because, by default, the signal has no effect.

Synchronous Signals

Synchronous signals are the result of an event that occurs inside a process and are delivered synchronously with respect to that event. For example, if a floating-point calculation results in an overflow, then a **SIGFPE** (floating-point exception signal) is delivered to the process immediately following the instruction that resulted in the overflow.

The default behavior of DCE Threads in DCE Version 1.0.2 when a synchronous terminating signal occurs is to dump core; that is, to not handle the signal. This differs from the behavior prior to DCE Version 1.0.2, in which such a signal would be turned into an exception and propagated out to whatever process was the original owner of the thread (namely the client, even though the exception might have occurred in the server). Therefore, if an application using DCE Threads wants to handle such signals, it must now set up a signal handler to do so by calling **sigaction()**. Note that the new DCE Threads behavior is in fact similar to the default behavior of most UNIX programs.

Synchronous, terminating signals represent an error that has occurred in the currently executing thread.

Asynchronous Signals

Asynchronous signals are the result of an event that is external to the process and are delivered at any point in a thread’s execution when such an event occurs. For example, when a user running a program types the interrupt character at the terminal (generally **<Ctrl-C>**), a **SIGINT** (interrupt signal) is delivered to the process.

Asynchronous, terminating signals represent an occurrence of an event that is external to the process and, if unhandled, results in the termination of the process. When an asynchronous terminating signal is delivered, DCE Threads catches it and checks to see if any threads are waiting for it. If threads are waiting, they are awakened, and the signal is considered handled and is dismissed. If there are no waiting threads, then DCE Threads causes the process to be terminated as if the signal had not been handled.

DCE Threads Signal Handling

DCE Threads provides the POSIX **sigwait()** service to allow threads to perform activities similar to signal handling without having to deal with signals directly. It also provides a jacket for **sigaction()** that allows each thread to have its own handler for synchronous signals.

In order to provide these mechanisms, DCE Threads installs signal handlers for most of the UNIX signals during initialization.

DCE Threads do not provide handlers for several UNIX signals. Those signals and the reasons why handlers are not provided are shown in Table 6.

Table 6. Signals for Which Handlers Are Not Provided

Signal	Reason Handler Is Not Provided
SIGKILL and **SIGSTOP**	These signals cannot be caught by user mode code.
SIGTRAP	Catching this signal interferes with debugging.
SIGTSTP and **SIGQUIT**	These signals are caught only while a thread has issued a **sigwait()** call because their default actions are otherwise valuable.

The POSIX sigwait() Service

The DCE Threads implementation of the POSIX **sigwait()** service allows any thread to block until one of a specified set of signals is delivered. A thread waits for any of the asynchronous signals, except for **SIGKILL** and **SIGSTOP**.

A thread cannot wait for a synchronous signal. This is because synchronous signals are the result of an error during the execution of a thread; if the thread is waiting for a signal, then it is not executing. Therefore, a synchronous signal cannot occur for a particular thread while it is waiting, and so the thread waits forever. POSIX stipulates that the thread must block the signals (using the UNIX system service **sigprocmask()**) it waits for before calling **sigwait()**.

The POSIX sigaction() Service

The DCE Threads implementation of the POSIX **sigaction()** service allows for per-thread handlers to be installed for catching synchronous signals. The **sigaction()** routine modifies behavior only for individual threads and works only for synchronous signals. Setting the signal action to **SIG_DFL** for a specific signal will restore the thread’s default behavior for that signal. Attempting to set a signal action for an asynchronous signal is an error.

The itimer VTALARM

DCE Threads installs a handler for the **itimer VTALARM**. Therefore, **VTALARM** is unavailable for use by other applications.

Alternatives to Using Signals

Avoid using UNIX signals in multithreaded programs. DCE Threads provides alternatives to signal handling. These alternatives are discussed in more detail in “Using Synchronization Objects” on page 129 and “Signaling a Condition Variable” on page 130.

Note: In order to implement these alternatives, DCE Threads must install its own signal handlers. These are installed when DCE Threads initializes itself, typically on the first thread-function call. At this time, any existing signal handlers are replaced.

Following are several reasons for avoiding signals:

- They cannot be used in a modular way in a multithreaded program.
- They are unnecessary when used as an asynchronous programming technique in a multithreaded program.
- There are almost no threads services available at signal level.
- There is no reliable, portable way to modify predicates.
- The signal-handler interface is unsuitable for use with threads. (For example, there is one signal action per signal per process, there is one signal mask per process, and **sigpause()** blocks the whole process.)

In a multithreaded program, signals cannot be used in a modular way because, on most current UNIX implementations, signals are inherently a process construct. There is only one instantiation of each signal and of each signal handler routine for all of the threads in an application. If one thread handles a particular signal in one way, and a different thread handles the same signal in a different way, then the thread that installs its signal handler last handles the signal. This applies only to asynchronously generated signals; synchronous signals can be handled on a per-thread basis using the DCE Threads **sigaction()** jacket.

Do not use asynchronous programming techniques in conjunction with threads, particularly those that increase parallelism such as using timer signals and I/O signals. These techniques can be complicated. They are also unnecessary because threads provide a mechanism for parallel execution that is simpler and less prone to error where concurrence can be of value. Furthermore, most of the threads routines are not supported for use in interrupt routines (such as signal handlers), and portions of runtime libraries cannot be used reliably inside a signal handler.

Nonthreaded Libraries

As programming with threads becomes common practice, you need to ensure that threaded code and nonthreaded code (code that is not designed to work with threads) work properly together in the same application. For example, you may write a new application that uses threads (for example, an RPC server), and link it with a library that does not use threads (and is thus not thread-safe). In such a situation you can do one of the following:

- Work with the nonthreaded software.
- Change the nonthreaded software to be thread-safe.

Working with Nonthreaded Software

Thread-safe code is code that works properly in a threaded environment. To work with nonthread-safe code, associate the global lock with all calls to such code.

You can implement the lock on the side of the routine user or the routine provider. For example, you can implement the lock on the side of the routine user if you write a new application like an RPC server that uses threads, and you link it with a library that does not. Or, if you have access to the nonthreaded code, the locks can be placed on the side of the routine provider, within the actual routine. Implement the locks as follows:

1. Associate one lock, a global lock, with execution of such code.
2. Require all of your threads to lock prior to execution of nonthreaded code.
3. Perform an unlock when execution is complete.

By using the global lock, you ensure that only one thread executes in outside libraries, which may call each other, and in unknown code. Using a single global lock is safer than using multiple local locks because it is difficult to be aware of everything a library may be doing or of the interactions that library can have with other libraries.

Making Nonthreaded Code Thread-Reentrant

Thread-reentrant code is code that works properly while multiple threads execute it concurrently. Thread-reentrant code is thread-safe, but thread-safe code may not be thread-reentrant. Document your code as being thread-safe or thread-reentrant.

More work is involved in making code thread-reentrant than in making code thread-safe. To make code thread-reentrant, do the following:

1. Use proper locking protocols to access global or static variables.
2. Use proper locking protocols when you use code that is not thread-safe.
3. Store thread-specific data on the stack or heap.
4. Ensure that the compiler produces thread-reentrant code.
5. Document your code as being thread-reentrant.

Avoiding Nonreentrant Software

The following subsections discuss two methods to help you avoid the pitfalls of nonreentrant software. These methods are as follows:

- Global lock
- Thread-specific storage

Global Lock

Use a global lock, which has the characteristics of a recursive mutex, instead of a regular mutex when calling routines that you think are nonreentrant. (When in doubt, assume the code is nonreentrant.)

The **pthread_lock_global_np()** routine is a locking protocol that is used to call nonreentrant routines, often found in existing library packages that were not designed to run in a multithreaded environment.

The way to call a library function that is not reentrant from a multithreaded program is to protect the function with a mutex. If every function that calls a library locks a particular mutex before the call and releases the mutex after the call, then the function completes without interference. However, this is difficult to do successfully because the function may be called by many libraries. A global lock solves this problem by providing a universal lock. Any code that calls any nonreentrant function uses the same lock.

To lock a global lock, call the **pthread_lock_global_np()** routine. To unlock a global lock, call the **pthread_unlock_global_np()** routine.

Note: Many COBOL and FORTRAN compilers generate inherently nonreentrant code. Many C, Ada, Pascal, and BLISS compilers generate reentrant code by default. It is possible to write nonreentrant code in the reentrant languages by not following a locking protocol.

Thread-Specific Storage

To avoid nonreentrancy when writing new software, avoid using global variables to store data that is thread-specific data.

Alternatively, allocate thread-specific data on the stack or heap and explicitly pass its address to called routines.

Avoiding Priority Inversion

Priority inversion occurs when interaction among three or more threads blocks the highest-priority thread from executing. For example, a high-priority thread waits for a resource locked by a low-priority thread, and the low-priority thread waits while a middle-priority thread executes. The high-priority thread is made to wait while a thread of lower priority (the middle-priority thread) executes.

To avoid priority inversion, associate a priority with each resource and force any thread using that object to first raise its priority to that associated with the object. This method of avoiding priority inversion is not a complete solution because all threads will then block at the same ceiling priority and be unblocked in FIFO order rather than by their actual priority.

The **SCHED_OTHER** (default) scheduling policy prevents priority inversion from causing a complete blockage of the high-priority thread because the low-priority thread is permitted to execute and release the resource. The **SCHED_FIFO** and **SCHED_RR** policies, however, do not force resumption of the low-priority thread if the middle-priority thread executes indefinitely.

Using Synchronization Objects

The following subsections discuss the use of mutexes to prevent two potential problems: race conditions and deadlocks. Also discussed is why you should signal a condition variable with the associated mutex locked.

Race Conditions

A race condition occurs when two or more threads perform an operation, and the result of the operation depends on unpredictable timing factors; specifically, when each thread executes and waits and when each thread completes the operation.

An example of a race condition is as follows:

1. Both A and B are executing (X = X + 1).
2. A reads the value of X (for example, X = 5).
3. B comes in and reads the value of X and increments it (making X = 6).

4. A gets rescheduled and now increments X. Based on its earlier read operation, A thinks (X+1 = 5+1 = 6). X is now 6. It should be 7 because it was incremented once by A and once by B.

To avoid race conditions, ensure that any variable modified by more than one thread has only one mutex associated with it. Do not assume that a simple add operation can be completed without allowing another thread to execute. Such operations are generally not portable, especially to multiprocessor systems. If it is possible for two threads to share a data point, use a mutex.

Deadlocks

A deadlock occurs when one or more threads are permanently blocked from executing because each thread waits on a resource held by another thread in the deadlock. A thread can also deadlock on itself.

The following is one technique for avoiding deadlocks:

1. Associate a sequence number with each mutex.
2. Lock mutexes in sequence.
3. Do not attempt to lock a mutex with a sequence number lower than that of a mutex the thread already holds.

Another technique, which is useful when a thread needs to lock the same mutex more than once before unlocking it, is to use a recursive mutex. This technique prevents a thread from deadlocking on itself.

Signaling a Condition Variable

When you are signaling a condition variable and that signal may cause the condition variable to be deleted, it is recommended that you signal or broadcast with the mutex locked.

The recommended coding for signaling a condition variable appears at the end of this chapter. The following two C code fragments show coding that is *not recommended*. The following C code fragment is executed by a releasing thread:

```
pthread_mutex_lock (m);
/* Change shared variables to allow */
/* another thread to proceed */

pthread_mutex_unlock (m); <---- Point A
pthread_cond_signal (cv); <---- Statement 1
```

The following C code fragment is executed by a potentially blocking thread:

```
pthread_mutex_lock (m);
while (!predicate ...
pthread_cond_wait (cv, m);

pthread_mutex_unlock (m);
```

Note: It is possible for a potentially blocking thread to be running at *Point A* while another thread is interrupted. The potentially blocking thread can then see the predicate true and therefore not become blocked on the condition variable.

Signaling a condition variable without first locking a mutex is not a problem. However, if the released thread deletes the condition variable without any further synchronization at *Point A*, then the releasing thread will fail when it attempts to execute *Statement 1* because the condition variable no longer exists.

This problem occurs when the releasing thread is a worker thread and the waiting thread is the boss thread, and the last worker thread tells the boss thread to delete the variables that are being shared by boss and worker.

The following C code fragment shows the *recommended* coding for signaling a condition variable while the mutex is locked:

```
pthread_mutex_lock (m);
/* Change shared variables to allow */
/* some other thread to proceed */

pthread_cond_signal (cv);  <---- Statement 1
pthread_mutex_unlock (m);
```

Chapter 9. Using the DCE Threads Exception-Returning Interface

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

- The routine returns a status value to the thread.
- The routine raises an exception.

Before you write a multithreaded program, you must choose only one of the preceding two methods of receiving status. These two methods cannot be used together in the same code module.

The POSIX P1003.4a (pthreads) draft standard specifies that errors be reported to the thread by setting the external variable **errno** to an error code and returning a function value of −1. The threads reference pages document this status-value-returning interface (see the *OSF DCE Application Development Reference*). However, an alternative to status values is provided by DCE Threads in the exception-returning interface.

This chapter introduces and provides conventions for the modular use of the exception-returning interface to DCE Threads.

Syntax for C

Access to exceptions from the C language is defined by the macros in the **exc_handling.h** file. The **exc_handling.h** header file is included automatically when you include **pthread_exc.h** (see “Invoking the Exception-Returning Interface” on page 135).

The following example shows the syntax for handling exceptions:

```
TRY
  try_block
[CATCH (exception_name)
  handler_block]...
[CATCH_ALL
  handler_block]
ENDTRY
```

A **try_block** or a **handler_block** is a sequence of statements, the first of which may be declarations, as in a normal block. If an exception is raised in the **try_block**, the catch clauses are evaluated in order to see if any one matches the current exception.

The **CATCH** or **CATCH_ALL** clauses absorb an exception; that is, they catch an exception propagating out of the **try_block**, and direct execution into the associated **handler_block**. Propagation of the exception, by default, then ends. Within the lexical scope of a handler, it is possible to explicitly cause propagation of the same exception to resume (this is called *reraising* the exception), or it is possible to raise some new exception.

The **RERAISE** statement is allowed in any handler statements and causes the current exception to be reraised. Propagation of the caught exception resumes.

The **RAISE** (*exception_name*) statement is allowed anywhere and causes a particular exception to start propagating. For example:

```
TRY
  sort(); /* Call a function that may raise an exception.
       * An exception is accomplished by longjumping
       * out of some nested routine back to the TRY
       * clause. Any output parameters or return
       * values of the called routine are therefore
       * indeterminate.
       */

CATCH (pthread_cancel_e)
  printf("Alerted while sorting\n"); RERAISE;

CATCH_ALL
  printf("Some other exception while sorting\n"); RERAISE;

ENDTRY
```

In the preceding example, if the **pthread_cancel_e** exception propagates out of the function call, the first **printf** is executed. If any other exception propagates out of sort, the second **printf** is executed. In either situation, propagation of the exception resumes because of the **RERAISE** statement. (If the code is unable to fully recover from the error, or does not understand the error, it needs to do what it did in the previous example and further propagate the error to its callers.)

The following shows the syntax for an epilogue:

```
TRY    try_block
[FINALLY final_block]
ENDTRY
```

The **final_block** is executed whether the **try_block** executes to completion without raising an exception, or if an exception is raised in the **try_block**. If an exception is raised in the **try_block**, propagation of the exception is resumed after executing the **final_block**.

Note that a **CATCH_ALL** handler and **RERAISE** could be used to do this, but the epilogue code would then have to be duplicated in two places, as follows:

```
TRY
   try_block
CATCH_ALL
   final_block
   RERAISE;
ENDTRY
{ final_block }
```

A **FINALLY** statement has exactly this meaning, but avoids code duplication.

Note: The behavior of **FINALLY** along with the **CATCH** or **CATCH_ALL** clauses is undefined. Do *not* combine them for the same **try_block**.

Another example of the **FINALLY** statement is as follows:

```
pthread__mutex_lock (some_object.mutex);
some_object.num_waiters = some_object.num_waiters + 1;
TRY
  while (! some_object.data_available)
    pthread_cond_wait (some_object.condition);
  /* The code to act on the data_available goes here */
```

```
FINALLY
  some_object.num_waiters = some_object.num_waiters - 1;
  pthread_mutex_unlock (some_object.mutex);
ENDTRY
```

In the preceding example, the call to **pthread_cond_wait()** could raise the **pthread_cancel_e** exception. The **final_block** ensures that the shared data associated with the lock is correct for the next thread that acquires the mutex.

Invoking the Exception-Returning Interface

To use the exception-returning interface, replace the first statement that follows with the second:

```
#include <pthread.h>
```

```
#include <pthread_exc.h>
```

Operations on Exceptions

An exception is an object that describes an error condition. Operations on exception objects allow errors to be reported and handled. If an exception is handled properly, the program can recover from errors. For example, if an exception is raised from a parity error while reading a tape, the recovery action may be to retry 100 times before giving up.

The DCE Threads exception-returning interface allows you to perform the following operations on exceptions:

- Declare and initialize an exception object
- Raise an exception
- Define a region of code over which exceptions are caught
- Catch a particular exception or all exceptions
- Define epilogue actions for a block
- Import a system-defined error status into the program as an exception

These operations are discussed in the following subsections.

Declaring and Initializing an Exception Object

Declaring and initializing an exception object documents that a program reports or handles a particular error. Having the error expressed as an exception object provides future extensibility as well as portability. Following is an example of declaring and initializing an exception object:

```
EXCEPTION parity_error;     /* Declare it */
EXCEPTION_INIT (parity_error); /* Initialize it */
```

Raising an Exception

Raising an exception reports an error, not by returning a value, but by propagating an exception. Propagation involves searching all active scopes for code written to handle the error or code written to perform scope-completion actions in case of any error, and then causing that code to execute. If a scope does not define a handler or epilogue block, then the scope is simply torn down as the exception propagates through the stack. This is sometimes referred to as *unwinding the stack*. DCE Threads exceptions are terminating; there is no option to make execution resume at the point of the error. (Execution resumes at the point where the exception was caught.)

If an exception is unhandled, the entire application process is terminated. Aborting the process, rather than just the faulting thread, provides clean termination at the point of error. This prevents the disappearance of the faulting thread from causing problems at some later point.

An example of raising an exception is as follows:

```
RAISE (parity_error);
```

Defining a Region of Code over Which Exceptions Are Caught

Defining a region of code over which exceptions are caught allows you to call functions that can raise an exception and specify the recovery action.

Following is an example of defining an exception-handling region (without indicating any recovery actions):

```
TRY {
  read_tape ();
  }
ENDTRY;
```

Catching a Particular Exception or All Exceptions

It is possible to discriminate among errors and perform different actions for each error.

Following is an example of catching a particular exception and specifying the recovery action (in this case, a message). The exception is reraised (passed to its callers) after catching the exception and executing the recovery action:

```
TRY {
  read_tape ();
  }
CATCH (parity_error) {
  printf ("Oops, parity error, program terminating\n");
  printf ("Try cleaning the heads!\n");
  RERAISE;
  }
ENDTRY
```

Defining Epilogue Actions for a Block

A **FINALLY** mechanism is provided so that multithreaded programs can restore invariants as certain scopes are unwound; for example, restoring shared data to a correct state and releasing locks. This is often the ideal way to define, in one place, the cleanup activities for normal or abnormal exit from a block that has changed some invariant.

Following is an example of specifying an invariant action whether or not there is an error:

```
lock_tape_drive (t);
TRY
  TRY
    read_tape ();
  CATCH (parity_error)
    printf ("Oops, parity error, program terminating\n");
    printf ("Try cleaning the heads!\n");
    RERAISE;
  ENDTRY
  /* Control gets here only if no exception is raised */
  /* ... Now we can use the data off the tape */
FINALLY
  /* Control gets here normally, or if any exception is */
  raised unlock_tape_drive (t);
ENDTRY
```

Importing a System-Defined Error Status into the Program as an Exception

Most systems define error messages by integer-sized status values. Each status value corresponds to some error message text that should be expressed in the user's own language. The capability to import a status value as an exception permits the DCE Threads exception-returning interface to raise or handle system-defined errors as well as programmer-defined exceptions.

An example of importing an error status into an exception is as follows:

```
exc_set_status (&parity_error, EPARITY);
```

The **parity_error** exception can then be raised and handled like any other exception.

Rules and Conventions for Modular Use of Exceptions

The following rules ensure that exceptions are used in a modular way so that independent software components can be written without requiring knowledge of each other:

- Use unique names for exceptions.

  A naming convention makes sure that the names for exceptions that are declared **EXTERN** from different modules do not clash. The following convention is recommended:

  ```
  <facility-prefix>_<error_name>_e
  ```

  For example, **pthread_cancel_e**.
- Avoid putting code in a **TRY** routine that belongs before it.

  The **TRY** only guards statements for which the statements in the **FINALLY**, **CATCH**, or **CATCH_ALL** clauses are always valid.

  A common misuse of **TRY** is to put code in the **try_block** that needs to be placed before **TRY**. An example of this misuse is as follows:

  ```
  TRY
    handle = open_file (file_name);
    /* Statements that may raise an exception here */
  FINALLY
    close (handle);
  ENDTRY
  ```

The preceding **FINALLY** code assumes that no exception is raised by **open_file**. This is because the code accesses an invalid identifier in the **FINALLY** part if **open_file** is modified to raise an exception. The preceding example needs to be rewritten as follows:

```
handle = open_file (file_name);
TRY
  {
  /* Statements that may raise an exception here */
  }
FINALLY
  close (handle);
ENDTRY
```

The code that opens the file belongs prior to **TRY**, and the code that closes the file belongs in the **FINALLY** statement. (If **open_file** raises exceptions, it may need a separate **try_block**.)

- Raise exceptions to their proper scope.

  Write functions that propagate exceptions to their callers so that the function does not modify any persistent process state before raising the exception. A call to the matching **close** call is required only if the **open_file** operation is successful in the current scope.

  If **open_file** raises an exception, the identifier will not be written, so **open_file** must not require that **close** be called when **open_file** raises an exception; that is, **open_file** should not be part of the **TRY** clause because that means **close** is called if **open_file** fails, and you cannot close an unopened file.

- Do not place a **RETURN** or nonlocal **GOTO** between **TRY** and **ENDTRY**.

  It is invalid to use **RETURN** or **GOTO**, or to leave by any other means, a **TRY**, **CATCH**, **CATCH_ALL**, or **FINALLY** block. Special code is generated by the **ENDTRY** macro, and it must be executed.

- Use the ANSI C volatile attribute.

  Variables that are read or written by exception-handling code must be declared with the ANSI C volatile attribute. Run your tests with the optimize compiler option to ensure that the compiler thoroughly tests your exception-handling code.

- Reraise exceptions that are not fully handled.

  You need to reraise any exception that you catch, unless your handler performs the complete recovery action for the error. This rule permits an unhandled exception to propagate to some final default handler that prints an error message to terminate the offending thread. (An unhandled exception is an exception for which recovery is incomplete.)

  A corollary of this rule is that **CATCH_ALL** handlers must reraise the exception because they may catch any exception, and usually cannot do recovery actions that are proper for every exception.

  Following this convention is important so that you also do not absorb a cancel or thread-exit request. These are mapped into exceptions so that exception handling has the full power to handle all exceptional conditions from access violations to thread exit. (In some applications, it is important to be able to catch these to work around an erroneously written library package, for example, or to provide a fully fault-tolerant thread.)

- Declare only static exceptions.

  For compatibility with C++, you need to only declare static exceptions.

DCE Threads Exceptions and Definitions

Table 7 lists the DCE Threads exceptions and briefly explains the meaning of each exception. Exception names beginning with **pthread_** are raised as the result of something happening internal to the DCE Threads facility and are not meant to be raised by your code. Exceptions beginning with **exc_** are generic and belong to the exception facility, the underlying system, or both. The pthread-specific extensions are listed followed by the generic extensions, each in alphabetical order.

Table 7. DCE Threads Exceptions

Exception	Definition
pthread_badparam_e	An improper parameter was used.
pthread_cancel_e	A thread cancellation is in progress.
pthread_defer_q_full_e	No space is currently available to process an interrupt request.
pthread_existence_e	The object referenced does not exist.
pthread_in_use_e	The object referenced is already in use.
pthread_nostackmem_e	No space is currently available to create a new stack.
pthread_notstack_e	The current stack was not created by DCE Threads.
pthread_signal_q_full_e	Unable to process condition signal from interrupt level.
pthread_stackovf_e	An attempted stack overflow was detected.
pthread_unimp_e	This is an unimplemented feature.
pthread_use_error_e	The requested operation is improperly invoked.
exc_decovf_e	An unhandled decimal overflow trap exception occurred.
exc_exquota_e	The operation failed due to an insufficient quota.
exc_fltdiv_e	An unhandled floating-point division by zero trap exception occurred.
exc_fltovf_e	An unhandled floating-point overflow trap exception occurred.
exc_fltund_e	An unhandled floating-point underflow trap exception occurred.
exc_illaddr_e	The data or object could not be referenced.
exc_insfmem_e	There is insufficient virtual memory for the requested operation.
exc_intdiv_e	An unhandled integer divide by zero trap exception occurred.
exc_intovf_e	An unhandled integer overflow trap exception occurred.
exc_nopriv_e	There is insufficient privilege for the requested operation.
exc_privinst_e	An unhandled privileged instruction fault exception occurred.
exc_resaddr_e	An unhandled reserved addressing fault exception occurred.

Table 7. DCE Threads Exceptions (continued)

Exception	Definition
exc_resoper_e	An unhandled reserved operand fault exception occurred.
exc_SIGBUS_e	An unhandled bus error signal occurred.
exc_SIGEMT_e	An unhandled EMT trap signal occurred.
exc_SIGFPE_e	An unhandled floating-point exception signal occurred.
exc_SIGILL_e	An unhandled illegal instruction signal occurred.
exc_SIGIOT_e	An unhandled IOT trap signal occurred.
exc_SIGPIPE_e	An unhandled broken pipe signal occurred.
exc_SIGSEGV_e	An unhandled segmentation violation signal occurred.
exc_SIGSYS_e	An unhandled bad system call signal occurred.
exc_SIGTRAP_e	An unhandled trace or breakpoint trap signal occurred.
exc_SIGXCPU_e	An unhandled CPU time limit exceeded signal occurred.
exc_SIGXFSZ_e	An unhandled file-size limit exceeded signal occurred.
exc_subrng_e	An unhandled subscript out-of-range trap exception occurred.
exc_uninitexc_e	An uninitialized exception was raised.

Chapter 10. DCE Threads Example

The example in this chapter shows the use of DCE Threads in a C program that performs a prime number search. The program finds a specified number of prime numbers, then sorts and displays these numbers. Several threads participate in the search: each thread takes a number (the next one to be checked), sees if it is a prime, records it if it is prime, and then takes another number, and so on.

This program shows the work crew model of programming (see “Chapter 6. Introduction to Multithreaded Programming” on page 103). The workers (threads) increment a number (**current_num**) to get their next work assignment, which in this case is the same task as before, but with a different number to check for a prime. As a whole, the worker threads are responsible for finding a specified number of prime numbers, at which point their work is completed.

Details of Program Logic and Implementation

The number of workers to be used and the requested number of prime numbers to be found are defined constants. A macro is used to check for bad status (bad status returns a value of −1), and to print a given string and the associated error value upon bad status. Data to be accessed by all threads (mutexes, condition variables, and so forth) are declared as global items.

Worker threads execute the prime search routine, which begins by synchronizing with the boss (or parent) thread by using a predicate and a condition variable. Always enclose a condition wait in a predicate loop to prevent a thread from continuing if it receives a spurious wakeup. The lock associated with the condition variable must be held by the thread when the condition wait call is made. The lock is implicitly released within the condition wait call and acquired again when the thread resumes. The same mutex must be used for all operations performed on a specific condition variable.

After the parent sets the predicate and broadcasts, the workers begin finding prime numbers until canceled by a fellow worker who has found the last requested prime number. Upon each iteration, the workers increment the current number to be worked on and take the new value as their work item. A mutex is locked and unlocked around getting the next work item. The purpose of the mutex is to ensure the atomicity of this operation and the visibility of the new value across all threads. This type of locking protocol needs to be performed on all global data to ensure its visibility and protect its integrity.

Each worker thread then determines if its current work item (a number) is prime by trying to divide numbers into it. If the number proves to be nondivisible, it is put on the list of primes. Cancels are explicitly turned off while working with the list of primes in order to better control any cancels that do occur. The list and its current count are protected by locks, which also protect the cancellation process of all other worker threads upon finding the last requested prime. While still under the prime list lock, the current worker checks to see if it has found the last requested prime, and if so unsets a predicate and cancels all other worker threads. Cancels are then reenabled. The canceling thread falls out of the work loop as a result of the predicate that it unsets.

The parent thread's flow of execution is as follows: set up the environment, create worker threads, broadcast to them that they can start, join each thread as it finishes, and sort and print the list of primes.

- Setting up of the environment requires initializing mutexes and the one condition variable used in the example.
- Creation of worker threads is straightforward and utilizes the default attributes (**pthread_attr_default**). Note again that locking is performed around the predicate on which the condition variable wait loops. In this case, the locking is simply done for visibility and is not related to the broadcast function.
- As the parent joins each of the returning worker threads, it receives an exit value from them that indicates whether a thread exited normally or not. In this case the exit values on all but one of the worker threads are **−1**, indicating that they were canceled.
- The list is then sorted to ensure that the prime numbers are in order from lowest to highest.

The following pthread routines are used in this example:

- **pthread_cancel()**
- **pthread_cond_broadcast()**
- **pthread_cond_init()**
- **pthread_cond_wait()**
- **pthread_create()**
- **pthread_detach()**
- **pthread_exit()**
- **pthread_join()**
- **pthread_mutex_init()**
- **pthread_mutex_lock()**
- **pthread_mutex_unlock()**
- **pthread_setcancel()**
- **pthread_testcancel()**

DCE Threads Example Body

The following is the DCE Threads example.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
/*
 * Constants used by the example.
 */
#define   workers   5      /* Threads to perform prime check */
#define   request   110     /* Number of primes to find    */
/*
 * Macros
 */
#define check(status,string) \
if (status == -1) perror (string)
/*
 * Global data
 */
pthread_mutex_t prime_list;   /* Mutex for use in accessing the prime */
pthread_mutex_t current_mutex; /* Mutex associated with current number */
pthread_mutex_t cond_mutex;   /* Mutex used for ensuring CV integrity */
pthread_cond_t cond_var;    /* Condition variable for thread start */
```

```
int       current_num= -1;/* Next number to be checked, start odd */
int       thread_hold= 1; /* Number associated w/condition state */
int       count=0;    /* Prime numbers count;/index to primes */
int       primes[request];/* Store primes; synchronize access   */
pthread_t    threads[workers]; /* Array of worker threads      */
/*
 * Worker thread routine.
 *
 * Worker threads start with this routine, which begins with a condition
 * wait designed to synchronize the workers and the parent. Each worker
 * thread then takes a turn taking a number for which it will determine
 * whether or not it is prime.
 *
 */
void
prime_search (pthread_addr_t arg)
  {
  div_t  div_results;      /* DIV results: quot and rem    */
  int   numerator;       /* Used for determining primeness  */
  int   denominator;      /* Used for determining primeness  */
  int   cut_off;        /* Number being checked div 2    */
  int   notifiee;        /* Used during a cancellation    */
  int   prime;         /* Flag used to indicate primeness */
  int   my_number;       /* Worker thread identifier     */
  int   status;         /* Hold status from pthread calls  */
  int   not_done=1;       /* Work loop predicate       */
  my_number = (int)arg;
  /*
   * Synchronize threads and the parent using a condition variable,
   * for which the predicate (thread_hold) will be set by the parent.
   */
  status = pthread_mutex_lock (&cond_mutex);
  check(status,"1:Mutex_lock bad status\n");

  while (thread_hold) {
    status = pthread_cond_wait (&cond_var, &cond_mutex);
    check(status,"3:Cond_wait bad status\n");
    }

  status = pthread_mutex_unlock (&cond_mutex);
  check(status,"4:Mutex_unlock bad status\n");
  /*
   * Perform checks on ever larger integers until the requested
   * number of primes is found.
   */
  while (not_done) {

    /* cancellation point */
    pthread_testcancel ();
    /* Get next integer to be checked */
    status = pthread_mutex_lock (&current_mutex);
    check(status,"6:Mutex_lock bad status\n");

    current_num = current_num + 2;       /* Skip even numbers */
    numerator = current_num;

    status = pthread_mutex_unlock (&current_mutex);
    check(status,"9:Mutex_unlock bad status\n");

    /* Only need to divide in half of number to verify not prime */
    cut_off = numerator/2 + 1;
    prime = 1;

    /* Check for prime; exit if something evenly divides */
    for (denominator = 2; ((denominator < cut_off) && (prime));
                        denominator++) {
      prime = numerator % denominator;
      }
    if (prime != 0) {

      /* Explicitly turn off all cancels */
      pthread_setcancel(CANCEL_OFF);

      /*
       * Lock a mutex and add this prime number to the list. Also,
       * if this fulfills the request, cancel all other threads.
       */
      status = pthread_mutex_lock (&prime_list);
      check(status,"10:Mutex_lock bad status\n");

      if (count < request) {
        primes[count] = numerator;
        count++;
        }
      else if (count == request) {
        not_done = 0;
        count++;
        for (notifiee = 0; notifiee < workers; notifiee++) {
          if (notifiee != my_number) {
            status = pthread_cancel (threads[notifiee] );
            check(status,"12:Cancel bad status\n");
            }
          }
        }

      status = pthread_mutex_unlock (&prime_list);
      check(status,"13:Mutex_unlock bad status\n");
      /* Reenable cancels */
      pthread_setcancel(CANCEL_ON);
      }
    pthread_testcancel ();
    }
  pthread_exit (my_number);
  }
main()
  {
  int   worker_num;   /* Counter used when indexing workers  */
  int   exit_value;   /* Individual worker's return status  */
  int   list;      /* Used to print list of found primes  */
  int   status;     /* Hold status from pthread calls    */
  int   index1;     /* Used in sorting prime numbers    */
  int   index2;     /* Used in sorting prime numbers    */
  int   temp;      /* Used in a swap; part of sort     */
  int   not_done;    /* Indicates swap made in sort     */

  * Create mutexes
  */
 status = pthread_mutex_init (&prime_list, pthread_mutexattr_default);
 check(status,"15:Mutex_init bad status\n");
 status = pthread_mutex_init (&cond_mutex, pthread_mutexattr_default);
 check(status,"16:Mutex_init bad status\n");
 status = pthread_mutex_init (&current_mutex, pthread_mutexattr_default);
 check(status,"17:Mutex_init bad status\n");

  /*
   * Create condition variable
   */
  status = pthread_cond_init (&cond_var, pthread_condattr_default);
  check(status,"45:Cond_init bad status\n");

  /*
   * Create the worker threads.
   */
  for (worker_num = 0; worker_num < workers; worker_num++) {
```

```
  status = pthread_create (
    &threads[worker_num],
    pthread_attr_default,
    prime_search,
    (pthread_addr_t)worker_num);
  check(status,"19:Pthread_create bad status\n");
}
/*
 * Set the predicate thread_hold to zero, and broadcast on the
 * condition variable that the worker threads may proceed.
 */
status = pthread_mutex_lock (&cond_mutex);
check(status,"20:Mutex_lock bad status\n");

thread_hold = 0;

status = pthread_cond_broadcast (&cond_var);
check(status,"20.5:cond_broadcast bad status\n");

status = pthread_mutex_unlock (&cond_mutex);
check(status,"21:Mutex_unlock bad status\n");
/*
 * Join each of the worker threads inorder to obtain their
 * summation totals, and to ensure each has completed
 * successfully.
 *
 * Mark thread storage free to be reclaimed upon termination by
 * detaching it.
 */
for (worker_num = 0; worker_num < workers; worker_num++) {

  status = pthread_join (
    threads[worker_num],
    &exit_value);
  check(status,"23:Pthread_join bad status\n");

if (exit_value == worker_num) printf("thread terminated normally\n");

  status = pthread_detach (&threads[worker_num]);
  check(status,"25:Pthread_detach bad status\n");
  }
/*
* Take the list of prime numbers found by the worker threads and
* sort them from lowest value to highest. The worker threads work
* concurrently; there is no guarantee that the prime numbers
* will be found in order. Therefore, a sort is performed.
*/
not_done = 1;
for (index1 = 1; ((index1 < request) && (not_done)); index1++) {
  for (index2 = 0; index2 < index1; index2++) {
    if (primes[index1] < primes[index2]) {
      temp = primes[index2];
      primes[index2] = primes[index1];
      primes[index1] = temp;
      not_done = 0;
      }
    }
  }

/*
 * Print out the list of prime numbers that the worker threads
 * found.
 */
printf ("The list of %d primes follows:\n", request);
printf("%d",primes[0]);

for (list = 1; list < request; list++) {
  printf (", %d", primes[list]);
  }

printf ("\n");
}
```

Part 3. DCE Remote Procedure Call

Chapter 11. Developing a Simple RPC Application

This chapter first explains how to write an interface definition in the DCE RPC Interface Definition Language (IDL) and illustrates the basic features of IDL. As an example, we present an interface definition for **greet**, a very simple application that prints greetings from a client and a remote server. The remainder of the chapter describes how to develop, build, and run the **greet** client and server programs.

The *OSF DCE Application Development Guide—Introduction and Style Guide* describes how to develop a DCE application by using many of the features of DCE. The following chapters use the term remote procedure call application (RPC application) to mean essentially the same thing, except in this context an RPC application concentrates on the features of the RPC technology, glossing over other DCE issues such as security, threads, and messaging. Since the RPC mechanism is the root technology for all DCE applications, the basic development approach is the same.

The Remote Procedure Call Model

A remote procedure call executes a procedure located in a separate address space from the calling code. The RPC model is a well-tested, industry-wide framework for distributing applications. The RPC model is derived from the programming model of local procedure calls and takes advantage of the fact that every procedure contains a procedure declaration. The procedure declaration defines the interface between the calling code and the called procedure. The procedure declaration defines the call syntax and parameters of the procedure. All calls to a procedure must conform to the procedure declaration.

Applications that use remote procedure calls look and behave much like local applications. However, an RPC application is divided into two parts: a server, which offers one or more sets of remote procedures, and a client, which makes remote procedure calls to RPC servers. A server and its clients generally reside on separate systems and communicate over a network. RPC applications depend on the RPC runtime to control network communications for them. The DCE RPC runtime supports additional tasks, such as finding servers for clients and managing servers.

A distributes application uses dispersed computing resources such as CPUs, databases, devices, and services. The following are examples:

- A calendar-management application that allows authorized users to access the personal calendars of other users.
- A graphics application that processes data on central CPUs and displays the results on workstations.
- A manufacturing application that shares information about assembly components among design, inventory, scheduling, and accounting programs located on different computers.

DCE RPC meets the basic requirements of a distributed application, including

- Clients finding the appropriate servers
- Data conversion for operating in a heterogeneous environment
- Network communications

Distributed applications include tasks such as managing communications, finding servers, providing security, and so forth. A standalone distributed application needs to perform all of these tasks itself. Without a convenient mechanism for these distributed computing tasks, writing distributed applications is difficult, expensive, and error-prone.

DCE RPC software provides the code, called stubs, and the RPC runtime that perform distributed computing tasks for your applications. This code and the runtime libraries are linked with client and server application code to form an RPC application.

Table 8 shows the basic tasks for the client and server of a distributed application. Calling the procedure and executing the remote procedure, shown in bold text, are performed by your application code (just as in a local application) but here they are in the client and server address spaces. For the other tasks, some are performed automatically by the stubs and RPC runtime, while others are performed by the RPC runtime via API calls in your application.

Table 8. Basic Tasks of an RPC Application

	Client Tasks	**Server Tasks**
1.		Select network protocols.
2.		Register RPC interfaces.
3.		Register endpoints in endpoint map.
4.		Advertise RPC interfaces and objects in the namespace.
5.		Listen for calls.
6.	Find compatible servers that offer the procedures.	
7.	**Call the remote procedure**.	
8.	Establish a binding with the server.	
9.	Convert input arguments into network data.	
10.	Transmit arguments to the server’s runtime.	
11.		Receive a call.
12.		Disassemble network data and convert input arguments into local data.
13.		Locate and invoke the called procedure.
14.		**Execute the remote procedure**.
15.		Convert the output arguments and return value into network data.
16.		Transmit results to the client’s runtime.
17.	Receive results.	
18.	Disassemble network data and convert output arguments into local data.	
19.	Return results and control to calling code.	

RPC Application Code

An RPC server or client contains application code, one or more RPC stubs, and the RPC runtime. RPC application code is the code written for a specific RPC application by the application developer. Application code implements and calls remote procedures, and also calls any RPC runtime routines the application needs. An RPC stub is an interface-specific code module that uses an RPC interface to pass and receive arguments. A server and a client contain complementary stubs for each RPC interface they share. The DCE RPC runtime manages communications for RPC applications. In addition, the DCE RPC runtime supports an application programming interface (API) used by RPC application code to enable RPC applications to set up their communications, manipulate information about servers, and perform optional tasks such as remotely managing servers and accessing security information.

Figure 16 shows the relationship of application code, stubs, and the RPC runtime in the server and client portions of an RPC application. The arrows show the direction calls are made by pointing to the called code.

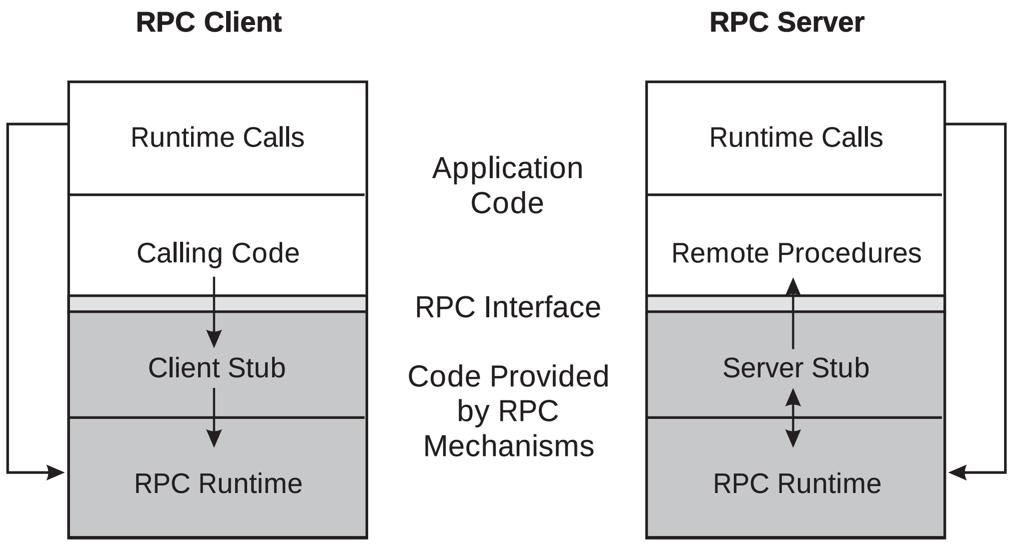


Figure 16. The Parts of an RPC Application

RPC application code differs for servers and clients. Minimally, server application code contains the remote procedures that implement one RPC interface, and the corresponding client contains calls to those remote procedures.

Stubs

The stub performs basic support functions for remote procedure calls. For instance, stubs prepare input and output arguments for transmission between systems with different forms of data representation. The stubs use the RPC runtime to handle the transmission between the client and server. The client stub can also use the runtime to find servers for the client.

When a client application calls a remote procedure, the client stub first prepares the input arguments for transmission. The process for preparing arguments for transmission is known as marshalling. Marshalling converts call arguments into a byte-stream format and packages them for transmission. Upon receiving call arguments, a stub unmarshalls them. Unmarshalling is the process by which a stub disassembles incoming network data and converts it into application data by using a format that the local system understands. Marshalling and unmarshalling both occur twice for each remote procedure call; that is, the client stub marshalls input arguments and unmarshalls output arguments, and the server stub unmarshalls input arguments and marshalls output arguments. Marshalling and unmarshalling permit client and server systems to use different data representations for equivalent data. For example, the client system can use ASCII characters and the server system can use EBCDIC characters, as shown in Figure 17.

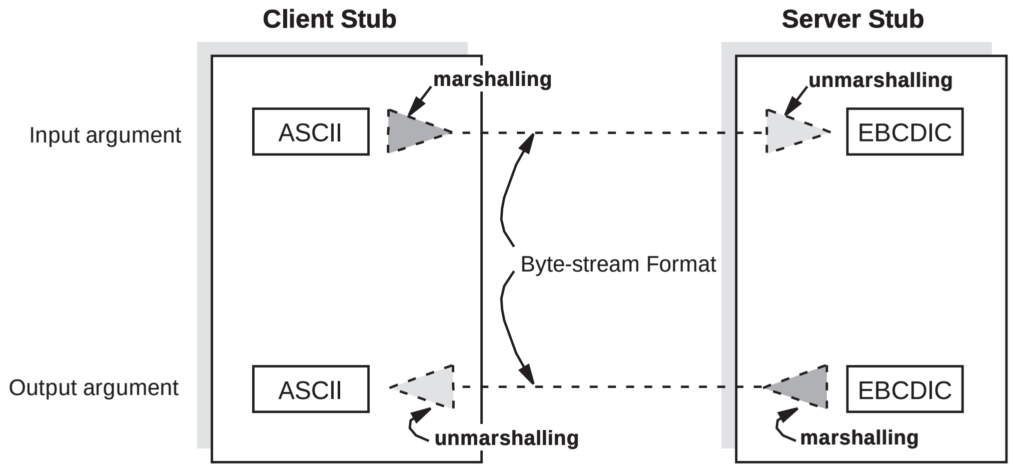


Figure 17. Marshalling and Unmarshalling Between ASCII and EBCDIC Data

The DCE IDL compiler (a tool for DCE application development) generates stubs by compiling an RPC interface definition written by application developers. The compiler generates marshalling and unmarshalling routines for platform-independent IDL data types.

To build the client for an RPC application, a developer links client application code with the client stubs of all the RPC interfaces the application uses. To build the server, the developer links the server application code with the corresponding server stubs.

The RPC Runtime

In addition to one or more RPC stubs, every RPC server and RPC client is linked with a copy of the RPC runtime. Runtime operations perform tasks such as controlling communications between clients and servers and finding servers for clients on request. An interface’s client and server stubs exchange arguments through their local RPC runtimes. The client runtime transmits remote procedure calls to the server. The server runtime receives the calls and dispatches each call to the appropriate server stub. The server runtime sends the call results to the client runtime. The DCE RPC runtime supports the RPC API used by RPC application code to call runtime routines.

Server application code must also contain server initialization code that calls RPC runtime routines when the server is starting up and shutting down. Client application code can also call RPC runtime routines. Server and client application code can also contain calls to RPC stub-support routines. Stub-support routines allow applications to manage programming tasks such as allocating and freeing memory.

RPC Application Components That Work Together

Figure 18 on page 153 shows the roles of application code, RPC stubs, and RPC runtimes during a remote procedure call.

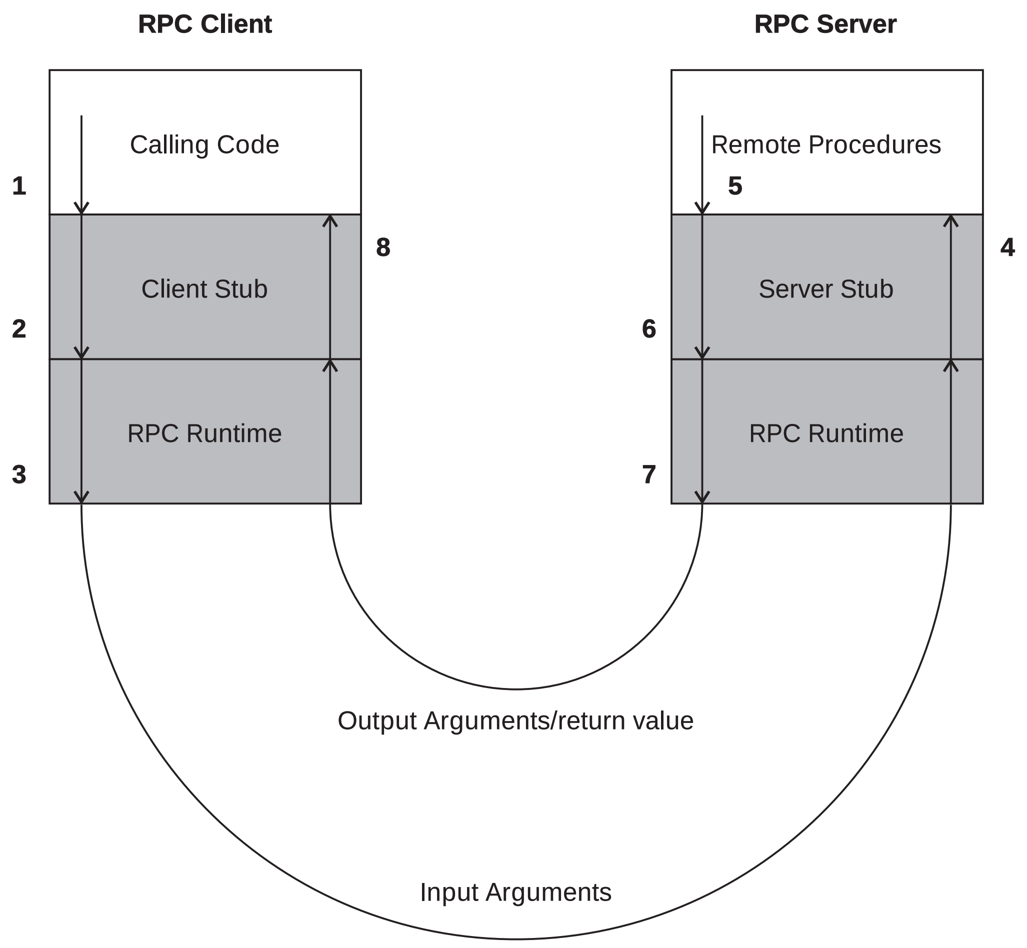


Figure 18. Interrelationships During a Remote Procedure Call

The following steps describe the interrelationships of the components of RPC applications, as shown in Figure 18:

1. The client's application code invokes a remote procedure call, passing the input arguments to the stub for the particular RPC interface.
2. The client's stub marshalls the input arguments and dispatches the call to the client's RPC runtime.
3. The client's RPC runtime transmits the input arguments to the server's RPC runtime, which dispatches the call to the server stub for the RPC interface of the called procedure.
4. The server's stub unmarshalls the input arguments and passes them to the called remote procedure.
5. The procedure executes and then returns any results (output arguments or a return value or both) to the server's stub.
6. The server's stub marshalls the results and returns them to the server's RPC runtime.
7. The server's RPC runtime transmits the results to the client's RPC runtime, which dispatches them to the correct client stub.
8. The client's stub unmarshalls output arguments and returns them to the calling code.

Overview of DCE RPC Development Tasks

The tasks involved in developing an RPC application resemble those involved in developing a local application. As an RPC developer, you perform the following basic tasks:

1. Design your application, deciding what procedures you need, which will be remote procedures, and how the remote procedures will be grouped into RPC interfaces.
2. Use the Universal Unique Identifier (UUID) generator to generate a UUID for each new interface.
3. Use the IDL to describe the RPC interfaces for the planned data types and remote procedures.
4. Use the DCE IDL compiler to generate the client and server stubs. (The IDL compiler can invoke the C compiler to create the stub object code.) Figure 19 illustrates this task.

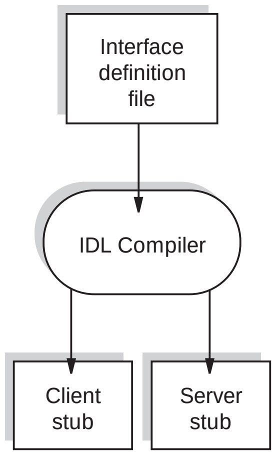


Figure 19. Generating Stubs

Note: Optionally, instead of generating stub object code (which is not portable), the IDL compiler can generate the stubs as ANSI C compliant source code.

5. Write or modify application code by using a compatible programming language, that is, a language that can be linked with C and can invoke C procedures, so the application code works with the stubs.

   Application code includes several kinds of code, as follows:

   a. Remote procedure calls
   b. Remote procedure implementations
   c. Initialization code (calls to RPC stub-support or runtime routines)
   d. Any non-RPC code your application requires
6. Generate object code from application code.
7. Create an executable client and server from the object files. Figure 20 on page 155 illustrates this task.

   For the client, link object code of the client stub(s) and the client application with the RPC runtime and any other needed runtime libraries.

For the server, link object code for the server stub(s), the initialization routines, and the set(s) of remote procedures with the RPC runtime and any other needed runtime libraries.

8. After initial testing, distribute the new application by separately installing the server and client executable images on systems on the network.

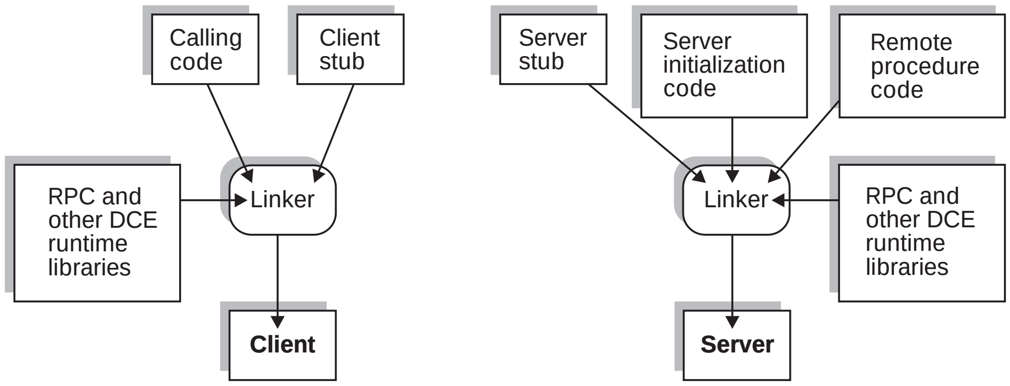

Calling code
Client stub
RPC and other DCE runtime libraries
Linker
Client
Server stub
Server initialization code
Remote procedure code
Linker
RPC and other DCE runtime libraries
Server

Figure 20. Building a Simple Client and Server

Writing an Interface Definition

Traditionally, calling code and called procedures share the same address space. In an RPC application, the calling code and the called remote procedures are not linked; rather, they communicate indirectly through an RPC interface. An RPC interface is a logical grouping of operations, data types, and constants that serves as a contract for a set of remote procedures. DCE RPC interfaces are compiled from formal interface definitions written by application developers using IDL.

The first step in developing a distributed application is to write an interface definition file in IDL. The IDL compiler, **idl**, uses the interface definition to generate a header file, a client stub file, and a server stub file. The IDL compiler produces header files in C and can produce stubs as C source files or as object files.

For some applications, you may also need to write an Attribute Configuration File (ACF) to accompany the interface definition. If an ACF exists, the IDL compiler interprets the ACF when it compiles the interface definition. Information in the ACF is used to modify the code that the compiler generates. (The **greet** example does not require an ACF.)

The remainder of this section briefly explains how to create an interface definition and gives simple examples of each kind of IDL declaration. For a detailed description of IDL, see “Chapter 18. Interface Definition Language” on page 357. For information on the IDL compiler, see the **idl(1rpc)** reference page.

An IDL interface definition consists of a header and a body. The following example shows the interface definition for the **greet** application:

```
/*
 * greet.idl
 *
 * The "greet" interface.
 */
```

```
[uuid(3d6ead56-06e3-11ca-8dd1-826901beabcd),
version(1.0)]

interface greetif
{
  const long int REPLY_SIZE = 100;

  void greet(
    [in]      handle_t h,
    [in, string]  char client_greeting[],
    [out, string]  char server_reply[REPLY_SIZE]
  );
}
```

The header of each RPC interface contains a UUID, which is a hexadecimal number that uniquely identifies an entity. A UUID that identifies an RPC interface is known as an interface UUID. The interface UUID ensures that the interface can be uniquely identified across all possible network configurations. In addition to an interface UUID, each RPC interface contains major and minor version numbers. Together, the interface UUID and version numbers form an interface identifier that identifies an instance of an RPC interface across systems and through time.

The interface body can contain the following constructs:

- Import declarations (not shown)
- Constant declarations (**REPLY_SIZE**)
- Data type declarations (not shown)
- Operation declarations (**void greet(...);**)

IDL declarations resemble declarations in ANSI C. IDL is purely a declarative language, so, in some ways, an IDL interface definition is like a C header file. However, an IDL interface definition must specify the extra information that is needed by the remote procedure call mechanism. Most of this information is expressed via IDL attributes. IDL attributes can apply to types, to type members, to operations, to operation parameters, or to an interface as a whole. An attribute is represented in **[ ]** (brackets) before the item to which it applies. In the **greet.idl** example, the **[in, string]** attributes associated with the **client_greeting** array means the parameter is for input only and that the array of characters has the properties of strings.

A comment can be inserted at any place in an interface definition where whitespace is permitted. IDL comments, like C comments, begin with **/*** (a slash and an asterisk) and end with ***/** (an asterisk and a slash).

RPC Interfaces That Represent Services

The simplest RPC application uses only one RPC interface. However, an application can use multiple RPC interfaces, and, frequently, an integral set of RPC interfaces work together as an RPC service. An RPC server is a logical grouping of one or more RPC interfaces. For example, you can write a calendar server that contains only a personal calendar interface or a calendar server that contains additional RPC interfaces such as a scheduling interface for meetings.

Different servers can share one or more RPC interfaces. For example, an administrative-support application can include an RPC interface from a calendar service.

An RPC interface exists independently of specific applications. Each RPC interface can be implemented by any set of procedures that conforms to the interface definition. The operations of an interface are exactly the same for all implementations of the same version of the interface. This makes it possible for clients from different implementations to call the same interface, and servers from different implementations to offer the same interface.

Figure 21 shows the role of RPC interfaces in remote procedure calls. This client contains calling code that makes two remote procedure calls. The first is a remote procedure call to Procedure A. The second is a remote procedure call to Procedure B.

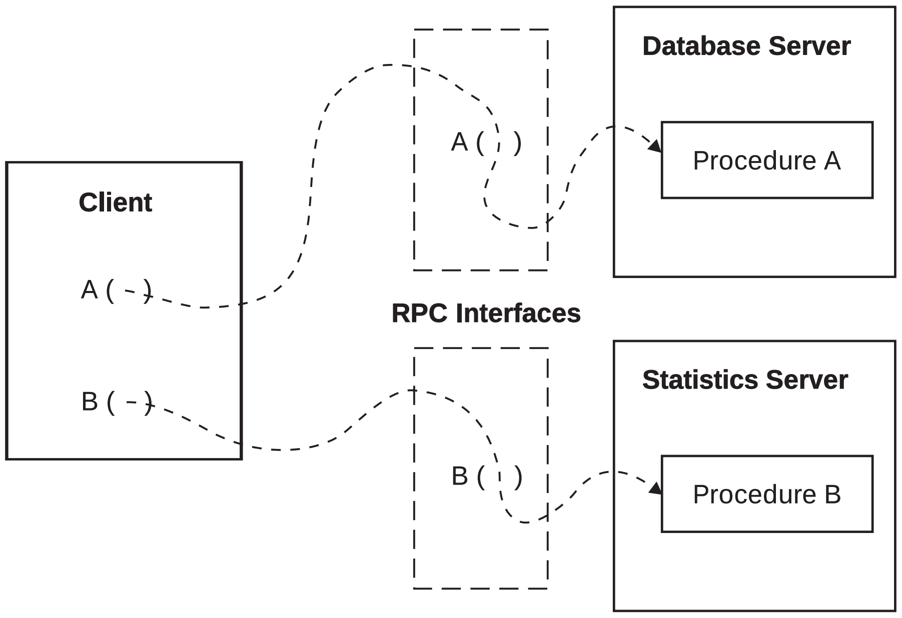


Figure 21. Role of RPC Interfaces

Clients can use any practical combination of RPC interfaces, whether offered by the same or different servers. For this example, using a database access interface, a client on a graphics workstation can call a remote procedure on a database server to fetch data from a central database. Then, using a statistics interface, the client can call a procedure on another server on a parallel processor to analyze the data from the central database and return the results to the client for display.

Generating an Interface UUID

The first step in building an RPC application is to generate a skeletal interface definition file and a UUID for the interface. Every interface in an RPC application must have a UUID. When you define a new interface, you must generate a new UUID for it.

Typically, you run **uuidgen** with the **-i** option, which produces a skeletal interface definition file and includes the generated UUID for the interface. For example, the following command creates a file **chess.idl**:

```
uuidgen -i > chess.idl
```

The contents of the file are as follows:

```
[
uuid(443f4b20-a100-11c9-baed-08001e0218cb),
version(1)
]
interface INTERFACENAME {

}
```

The first part of the skeletal definition is the header, which specifies a UUID, a version number, and a name for the interface. The last part of the definition is **{ }** (an empty pair of braces); import, constant, type, and operation declarations go between these braces.

By convention, the names of interface definition files end with the suffix **.idl**. The IDL compiler constructs names for its output files based on the interface definition filename and uses the following default suffixes:

- **.h** for header files
- **_cstub.o** for client stub files
- **_sstub.o** for server stub files

For example, compilation of a **chess.idl** interface definition file would produce a **chess.h** header file, a **chess_cstub.o** client stub file, and a **chess_sstub.o** server stub file. (The IDL compiler creates C language intermediate files and by default invokes the C compiler to produce object files, but it can also retain the C language files.)

For more information on the UUID generator, see the **uuidgen(1rpc)** reference page.

Naming the Interface

After you have used **uuidgen** to generate a skeletal interface definition, replace the dummy string **INTERFACENAME** with the name of your interface.

By convention, the name of an interface definition file is the same as the name of the interface it defines, with the suffix **.idl** appended. For example, the definition for a **bank** interface would reside in a **bank.idl** interface definition file, and, if the application required an ACF, its name would be **bank.acf**.

The IDL compiler incorporates the interface name in identifiers it constructs for various data structures and data types in the **.h** file, so the length of an interface name can be at most 17 characters. (Most IDL identifiers have a maximum length of 31 characters.)

Specifying Interface Attributes

Interface attributes are defined within **[ ]** (brackets) in the header of the interface definition. The definition for any remote interface needs to specify the **uuid** and **version** interface attributes, so these are included in the skeletal definition that **uuidgen** produces.

If an interface is exported by servers on well-known endpoints, these endpoints must be specified via the **endpoint** attribute. Interfaces that use dynamic endpoints do not have this attribute. (A well-known endpoint is a stable address on the host, while a dynamic endpoint is an address that the RPC runtime requests when the server is started.)

The interface definition language can be used to specify procedure prototypes for any application, even if the procedures are never used remotely. If all of the procedures of an interface are called only locally and never remotely, the interface can be given the **local** attribute. Since local calls do not have any network overhead, the **local** attribute causes the compiler to generate only a header file, not stubs, for the interface.

Import Declarations

The IDL **import** declaration specifies another interface definition whose types and constants are used by the importing interface. (Similar to the **include** declaration in C.)

The **import** declaration allows you to collect declarations for types and constants that are used by several interfaces into one common file. For example, if you are defining two database interfaces named **dblookup** and **dbupdate**, and these interfaces have many data types and constants in common, you can declare those data types and constants in a **dbcommon.idl** file and import this file in the **dblookup.idl** and**dbupdate.idl** interface definitions. For example:

```
import "dbcommon.idl";
```

By default, the IDL compiler resolves relative pathnames of imported files by looking first in the current working directory and then in the system IDL directory. The **-I** option of the IDL compiler allows you to specify additional directories to search. You can thereby avoid putting absolute pathnames in your interface definitions. For example, if an imported file has the absolute pathname **/dbproject/src/dbconstants.idl**, then the IDL compiler option **-I/dbproject/src** allows you to import the file by its leaf name, **dbconstants.idl**.

Constant Declarations

The IDL **const** declaration allows you to declare integer, Boolean, character, string, and null pointer constants, some of which are shown in the following examples:

```
const short TEN = 10;
const boolean VRAI = TRUE;
const char* JSB = "Johann Sebastian Bach";
```

Type Declarations

To support application development in a variety of languages and to support the special needs of distributed applications, IDL provides an extensive set of data types, including the following:
- Simple types, such as integers, floating-pointing numbers, characters, Booleans, and the primitive binding-handle type **handle_t** (usually equivalent to **rpc_binding_handle_t**)
- Predefined types, including ISO international character types and the error status type **error_status_t**
- Constructed types, such as strings, structures, unions, arrays, pointers, and pipes

The IDL **typedef** declaration lets you give a name to any types you construct.

The general form of a type declaration is

```
typedef [ type_attribute,...] type_specifier type_declarator,...;
```

where the bracketed list of type attributes is optional. The *type_specifier* specifies a simple type, a constructed type, a predefined type, or a type previously named in the interface. Each *type_declarator* is a name for the type being defined. As in C, arrays and pointers are declared by the *type_declarator* constructs **[ ]** (brackets) and ***** (an asterisk).

The following type declaration uses the IDL's simple data type, **long** (a 32-bit data type), to define the **integer32** integer type:

```
typedef long integer32;
```

The *type_specifier* constructs for structures and unions permit the application of attributes to members. In the following example, one member of a structure is a conformant array (an array without a fixed upper bound), and the **size_is** attribute names another member of the structure that at runtime provides information about the size of the array:

```
typedef struct {
   long dsize;
   [size_is(dsize)] float darray[ ];
   } dataset;
```

Operation Declarations

Operation declarations specify the signature of each operation in the interface, including the operation name, the type of data returned, and the types of all parameters passed (if any) in a call.

The general form of an operation declaration is

```
[operation_attribute, ...] type_specifier operation_identifier ([parameter_declaration,...]);
```

where the bracketed list of operation attributes is optional. Among the possible attributes of an operation are **idempotent**, **broadcast**, and **maybe**, which specify semantics to be applied by the RPC runtime mechanism when the operation is called. If an operation when called once can safely be executed more than once, the IDL declaration of the operation may specify the **idempotent** attribute; **idempotent** semantics allow remote procedure calls to execute more efficiently by letting the underlying RPC mechanism retry the procedure if it deems it necessary.

The *type_specifier* specifies the type of data returned by the operation.

The *operation_identifier* names the operation. Although operation names are arbitrary, a common convention is to use the name of an interface as a prefix for the names of its operations. For example, a **bank** interface may have operations named **bank_open**, **bank_close**, **bank_deposit**, **bank_withdraw**, and **bank_balance**.

Each *parameter_declaration* in an operation declaration declares a parameter of the operation. A *parameter_declaration* takes the following form:

```
[parameter_attribute, ...] type_specifier parameter_declarator
```

Every parameter attribute must have at least one of the parameter attributes **in** or **out** to specify whether the parameter is passed as an input, as an output, or in both directions. The *type_specifier* and *parameter_declarator* specify the type and name of the parameter.

Output parameters must be passed by reference and therefore must be declared as pointers via the pointer operator ***** (an asterisk) or an array.

If you want an interface to always use explicit binding handles, the first parameter of each operation declaration must be a binding handle, as in the following example:

```
void greet(
  [in]      handle_t h,
  [in, string]  char client_greeting[],
  [out, string]  char server_reply[REPLY_SIZE]
);
```

However, if you want applications to use the ACF feature of an implicit binding handle (or even an automatic binding handle) for some or all procedures, operation declarations must not have binding handle parameters in the interface definition:

```
void greet_no_handle(
  [in, string]  char client_greeting[],
  [out, string]  char server_reply[REPLY_SIZE]
);
```

This form of operation declaration is the most flexible because applications can always specify explicit, implicit, or automatic binding handles through an ACF.

Running the IDL Compiler

After you have written an interface definition, run the IDL compiler to generate header and stub files. The compiler offers many options that, for example, allow you to choose what C compiler or C preprocessor commands are run, what directories are searched for imported files, which of the possible output files are generated, and how the output files are named.

The **greet.idl** interface definition can be compiled by the following command:

```
idl greet.idl
```

This compilation produces a header file (**greet.h**), a client stub file (**greet_cstub.o**), and a server stub file (**greet_sstub.o**. For complete information on running the IDL compiler, see the **idl(1rpc)** reference page.

Writing the Client Code

This section describes the client program for the **greet** application, whose interface definition was shown earlier in this chapter.

The client performs the following major steps:

1. It checks the command-line arguments for an entry name to use for its search in the namespace.
2. It calls **rpc_ns_binding_import_begin()** to start the search in the namespace.
3. It calls **rpc_ns_binding_import_next()** to obtain a binding to a server.
4. It calls the **greet** remote procedure with a string greeting.
5. It prints the reply from the server.

The **greet_client.c** module is as follows:

```
/*
 * greet_client.c
 *
 * Client of "greet" interface.
 */
#include <stdio.h>
#include <dce/rpc.h>

#include "greet.h"
#include "util.h"

int
main(
  int argc,
  char *argv[]
)
{
  rpc_ns_handle_t import_context;
  handle_t binding_h;
  error_status_t status;
  idl_char reply[REPLY_SIZE];

  if (argc < 2) {
    fprintf(stderr, "usage: greet_client <CDS pathname>\n");
    exit(1);
  }
  /*
   * Start importing servers using the name specified
   * on the command line.
   */
  rpc_ns_binding_import_begin(
    rpc_c_ns_syntax_default, (unsigned_char_p_t) argv[1],
      greetif_v1_0_c_ifspec, NULL, &import_context, &status);
  ERROR_CHECK(status, "Can't begin import");
  /*
   * Import the first server (we could iterate here,
   * but we'll just take the first one).
   */
  rpc_ns_binding_import_next(import_context, &binding_h, &status);
  ERROR_CHECK(status, "Can't import");
  /*
   * Make the remote call.
   */
  greet(binding_h, (idl_char *) "hello, server", reply);

  printf("The Greet Server said: %s\n", reply);
}
```

The module first includes **greet.h**, the header file for the **greet** interface generated by the IDL compiler.

In this example, after each call to an RPC runtime routine, the client program calls the application-specific **ERROR_CHECK** macro. If the status from the RPC runtime routine is not **error_status_ok**, **dce_error_inq_text()** is called and the error message is printed.

As specified in the **greet.idl** interface definition, the **greet** application uses explicit handles. The client therefore passes a binding handle of type **handle_t** as the first parameter of the **greet** procedure. At runtime, when the client makes its first remote procedure call, the handle is only partially bound because the client does not know the particular endpoint on which the server is listening; for delivery of its requests to the server endpoint, the client depends on the endpoint mapping service of the **dced** process on the server host.

Writing the Server Code

The following subsections describe the server program for the **greet** application. The **greet_server** program takes one argument and is invoked as follows:

greet_server *CDS_pathname*

The **greet_server** program uses the input argument to establish an entry for itself in the DCE CDS namespace.

The **greet** server program has two user-written modules:

- The **greet_server.c** module contains the server **main** function and performs the initialization and registration required to export the **greet** interface.
- The **greet_manager.c** module contains the code that actually implements the **greet** operation.

The greet_server.c Source Code

Most applications should use the DCE convenience routines for server initialization routines (routines that begin with **dce_server_**) to prepare servers to listen for remote procedure calls. These routines are simple to use, prepare a server so that **dced** can manage it, and they allow enough flexibility to do most typical initializations. However, for detailed control, applications can also use the lower-level RPC API to do server initialization. In this chapter, we describe how to use the RPC API for server initialization.

In this section, the **greet_server.c** module is described and shown in successive pieces.

Including idl-Generated Headers

Like **greet_client.c**, the **greet_server.c** module includes **greet.h** so that constants, data types, and procedure prototypes are available in the application. For example:

```
/*
 * greet_server.c
 *
 * Main program (initialization) for "greet" server.
 */

#include <stdio.h>
#include <dce/dce_error.h>
#include <dce/rpc.h>
```

```
#include "greet.h"
#include "util.h"
int
main(
  int argc,
  char *argv[]
)
{
  unsigned32 status;
  rpc_binding_vector_t *binding_vector;

  if (argc < 2) {
    fprintf(stderr, "usage: greet_server <CDS pathname>\n");
    exit(1);
  }
```

Registering the Interface

The server calls **rpc_server_register_if()**, supplying its interface specifier (defined in **greet.h**), to register each interface with the RPC runtime:

```
/*
 * Register interface with RPC runtime.
 */
rpc_server_register_if(greetif_v1_0_s_ifspec, NULL, NULL,
  &status);
ERROR_CHECK(status, "Can't register interface");
```

Selecting Protocol Sequences

The server calls **rpc_server_use_all_protseqs()** to obtain endpoints on which to listen for remote procedure calls:

```
/*
 * Use all protocol sequences that are available.
 */
rpc_server_use_all_protseqs(rpc_c_protseq_max_reqs_default,
  &status);
ERROR_CHECK(status, "Can't use protocol sequences");
```

Obtaining the Server's Binding Handles

To obtain a vector of binding handles that it can use when registering endpoints, the server calls **rpc_server_inq_bindings()**:

```
/*
 * Get the binding handles generated by the runtime.
 */
rpc_server_inq_bindings(&binding_vector, &status);
ERROR_CHECK(status, "Can't get bindings for server");
```

Registering Endpoints

A call to **rpc_ep_register()** registers the server endpoints in the endpoint mapper service of the local **dced**:

```
/*
 * Register assigned endpoints with endpoint mapper.
 */
rpc_ep_register(
  greetif_v1_0_s_ifspec, binding_vector, NULL,
  (unsigned_char_p_t) "greet server version 1.0", &status);
ERROR_CHECK(status, "Can't register with endpoint map");
```

Exporting to CDS

To advertise itself to clients, the server calls **rpc_ns_binding_export()**. The server entry for the namespace is obtained from the argument input when the server is invoked (**argv[1]**).

```
/*
 * Export ourselves into the CDS namespace.
 */
rpc_ns_binding_export(
  rpc_c_ns_syntax_default, (unsigned_char_p_t) argv[1],
  greetif_v1_0_s_ifspec, binding_vector, NULL, &status);
ERROR_CHECK(status, "Can't export into CDS namespace");
```

Listening for Calls

To begin listening for remote procedure call requests, the server calls **rpc_server_listen()**.

```
/*
 * Start listening for calls.
 */
printf("Listening...\n");

rpc_server_listen(rpc_c_listen_max_calls_default, &status);
ERROR_CHECK(status, "Can't start listening for calls");
```

The greet_manager.c Source Code

The **greet_manager.c** module includes **greet.h** and it also defines the routine **greet**, as follows:

```
/*
 * greet_manager.c
 *
 * Implementation of "greet" interface.
 */

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

void
greet(
  handle_t h,
  idl_char *client_greeting,
  idl_char *server_reply
)
{
  printf("The client says: %s\n", client_greeting);

  strcpy(server_reply, "Hi, client!");
}
```

Building the greet Programs

The client side of the **greet** application is the **greet_client** program, which is built from the following:

- The user-written **greet_client.c** client module
- The IDL-compiler-generated **greet_cstub.o** client stub module
- The user-written **util.c** module containing the error-checking routine

- DCE libraries

The server side of the **greet** application is the **greet_server** program, which is built from the following:

- The user-written **greet_server.c** server module
- The user-written **greet_manager.c** manager module
- The user-written **util.c** module containing the error-checking routine
- The IDL-compiler-generated **greet_sstub.o** server stub module
- DCE libraries

These programs can be built by **make** with a makefile such as the following:

```
DCEROOT = /opt/dcelocal
CC = /bin/cc
IDL = idl
LIBDIRS = -L${DCEROOT}/usr/lib
LIBS = -ldce
LIBALL = ${LIBDIRS} ${LIBS}
INCDIRS = -I. -I${DCEROOT}/share/include
CFLAGS = -g ${INCDIRS}
IDLFLAGS = -v ${INCDIRS} -cc_cmd "${CC} ${CFLAGS} -c"

all: greet_client greet_server

greet.h greet_cstub.o greet_sstub.o: greet.idl
${IDL} ${IDLFLAGS} greet.idl

greet_client: greet.h greet_client.o util.o greet_cstub.o
${CC} -o greet_client greet_client.o greet_cstub.o util.o \
 ${LIBALL}

greet_server: greet.h greet_server.o greet_manager.o util.o \
 greet_sstub.o
${CC} -o greet_server greet_server.o greet_manager.o \
    greet_sstub.o util.o ${LIBALL}

greet_client.c greet_server.c util.c: util.h
greet_manager.c greet_client.c greet_server.c:
greet.h
```

Running the greet Programs

Running the **greet** application involves starting the server program and then running the client program. Before starting the server program, you need write access to the CDS namespace and you need to ensure that the **dced** process is running on the server host. For more information, see the **dced(8dce)** reference page.

You start the server program by using a CDS entry such as the following:

```
greet_server /.:/greet_entry
Listening...
```

You start the client on another host (or even the same host) by using the same CDS entry as follows:

```
greet_client /.:/greet_entry
```

The following message is printed on the server's host:

```
The client says: hello, server
```

The following reply is printed on the client's host:

```
The Greet Server said: Hi, client!
```

The server program can be terminated at any time by a signal, which on many systems can be generated by **<Ctrl-C>**.

When applications such as **greet** execute, many errors can occur that have nothing to do with your own code. In general, errors that occur when a remote procedure call executes are reported as exceptions. For example, exceptions that the client side of **greet_client** could raise if the server suddenly and unexpectedly halts include (but are not limited to) **rpc_x_comm_failure** and **rpc_x_call_timeout**. Other ways to respond to these errors are available through the **comm_status** and **fault_status** attributes in an interface definition or attribute configuration file. Explanations of these attributes appear in "Chapter 19. Attribute Configuration Language" on page 425. Also, see "Chapter 17. Topics in RPC Application Development" on page 313, which explains the guidelines for error handling.

In addition, "Part 2. DCE Threads" on page 101 of this guide contains information about the macros (such as those specified by **TRY**, **CATCH**, and **ENDTRY** statements) for exception handling. If an exception occurs that the client application does not handle, it causes the client to terminate with an error message. The client's termination could include a core dump or other system-dependent error-reporting method. Detailed explanations of RPC status codes and RPC exceptions are in the *OSF DCE Problem Determination Guide*.

Chapter 12. RPC Fundamentals

DCE RPC provides a call environment that behaves essentially like a local call environment. However, some special requirements are imposed on remote procedure calls by the remoteness of calling code to the called procedure. Therefore, a remote procedure call may not always behave exactly like a local procedure call.

This chapter discusses the following topics:

- Universal unique identifiers
- Communications protocols
- Binding information
- Endpoints
- Execution semantics
- Communication failures
- Scaling applications
- RPC Objects

Distributed applications have the following implications:

- Client/server relationship—binding

  Like a local procedure call, a remote procedure call depends on a static relationship between the calling code and the called procedure. In a local application, this relationship is established by linking the calling and called code. Linking gives the calling code access to the address of each procedure to be called. Enabling a remote procedure call to go to the right procedure requires a similar relationship (called a *binding*) between a client and a server. A binding is a temporary relationship that depends on a communications link between the client and server RPC runtimes. A client establishes a binding over a specific protocol sequence to a specific host system and endpoint.
- Independent address spaces

  The calling code and called remote procedure reside in different address spaces, generally on separate systems. The calling and called code cannot share global variables or other global program state such as open files. All data shared between the caller and the called remote procedure must be specified as procedure parameters in the IDL specification. Unlike a local procedure call that commonly uses the call-by-reference passing mechanism for input/output parameters, remote procedure calls with input/output parameters have copy-in/copy-out semantics due to the differing address spaces of the calling and called code. These two passing mechanisms are only slightly different, and most procedure calls are not sensitive to the differences between call-by-reference and copy-in/copy-out semantics.
- Independent failure

  Distributing a calling program and the called procedures to physically separate machines increases the complexity of procedure calls. Remoteness introduces issues such as a remote system crash, communications failures, naming and binding issues, security problems, and protocol incompatibilities. Such issues can require error handling that is unnecessary for local procedure calls. Also, as with local procedure calls, remote procedure calls are subject to execution errors that arise from the procedure call itself.

Universal Unique Identifiers

Each UUID contains information, including a timestamp and a host identifier. Applications use UUIDs to identify many kinds of entities. DCE RPC identifies several uses of UUIDs, according to the kind of entities each identifies:

- Interface UUID

  A UUID that identifies a specific RPC interface. An interface UUID is declared in an RPC interface definition (an IDL file) and is a required element of the interface. For example:

  ```
  uuid(2fac8900-31f8-11ca-b331-08002b13d56d),
  ```
- Object UUID

  A UUID that identifies an entity for an application; for example, a resource, a service, or a particular instance of a server. An application defines an RPC object by associating the object with its own UUID known as an object UUID. The object UUID exists independently of the object, unlike an interface UUID. A server usually generates UUIDs for its objects as part of initialization. A given object UUID is meaningful only while a server is offering the corresponding RPC object to clients.

  To distinguish a specific use of an object UUID, a UUID is sometimes labeled for the entity it identifies. For example, an object UUID that is used to identify a particular instance of a server is known as an instance UUID.
- Type UUID

  A UUID that identifies a set of RPC objects and an associated manager (the set of remote procedures that implements an RPC interface for objects of that type). This is often called a *manager type UUID*.

Servers can create object and type UUIDs by calling the **uuid_create()** routine.

Communications Protocols

A communications link depends on a set of communications protocols. A communications protocol is a clearly defined set of operational rules and procedures for communications.

Communications protocols include a transport protocol (from the Transport Layer of the OSI network architecture) such as the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP); and the corresponding network protocol (from the OSI Network Layer) such as the Internet Protocol (IP).

For an RPC client and server to communicate, their RPC runtimes must use at least one identical communications protocol, including a common RPC protocol, transport protocol, and network protocol. An RPC protocol is a communications protocol that supports the semantics of the DCE RPC API and runs over specific combinations of transport and network protocols. DCE RPC provides two RPC protocols: the connectionless RPC protocol and the connection-oriented RPC protocol.

- Connectionless (Datagram) RPC protocol

  This protocol runs over a connectionless transport protocol such as UDP. The connectionless protocol supports broadcast calls.
- Connection-oriented RPC protocol

  This protocol runs over a connection-oriented transport protocol such as TCP.

Each binding uses a single RPC protocol and a single pair of transport and network protocols. Only certain combinations of communications protocols are functionally valid (are actually useful for interoperation); for instance, the RPC connectionless protocol cannot run over connection-oriented transport protocols such as TCP. DCE RPC supports the following combinations of communications protocols (as provided by OSF):

- RPC connection-oriented protocol over the Internet Protocol Suite, Transmission Control Protocol (TCP/IP)
- RPC connectionless protocol over the Internet Protocol Suite, User Datagram Protocol (UDP/IP)

Binding Information

Binding information includes a set of information that identifies a server to a client or a client to a server. Each instance of binding information contains all or part of a single address. The RPC runtime maintains binding information for RPC servers and clients. To make a specific instance of locally maintained binding information available to a given server or client, the runtime creates a local reference known as a binding handle. Servers and clients use binding handles to refer to binding information in runtime calls or remote procedure calls. A server obtains a complete list of its binding handles from its RPC runtime. A client obtains one binding handle at a time from its RPC runtime. Figure 22 illustrates a binding.

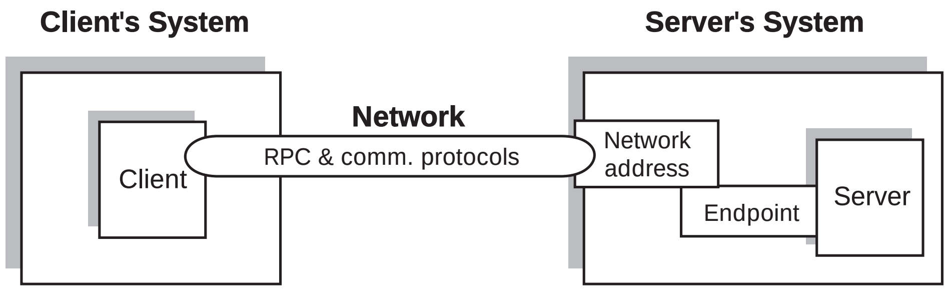


Figure 22. A Binding

Binding information includes the following components:

- Protocol sequence

  A valid combination of communications protocols presented by the runtime as a character string. Each protocol sequence includes a network protocol, a transport protocol, and an RPC protocol that works with them.

  An RPC server tells the runtime which protocol sequences to use when listening for calls to the server, and its binding information contains those protocol sequences.

- Network addressing information

  Includes the network address and the endpoint of a server.

  – The network address identifies a specific host system on a network. The format of the address depends on the network protocol portion of the protocol sequence.

  – The endpoint acts as the address of a specific server instance within the host system. The format of the endpoint depends on the transport protocol portion of the protocol sequence. For each protocol sequence a server instance uses, it requires a unique endpoint. A given endpoint can be used by only one server per system, assigned by the local system on a first-come, first-served basis.

- Transfer Syntax

  The server's RPC runtime must use a transfer syntax that matches one used by the client's RPC runtime. A transfer syntax is a set of encoding rules used for the network transmission of data and the conversion to and from different local data representations. A shared transfer syntax enables communications between systems that represent local data differently. DCE RPC currently uses a single transfer syntax, Network Data Representation (NDR). NDR encodes data into a byte stream for transmission over a network. A transfer syntax such as NDR enables machines with different formats to exchange data successfully. (The DCE RPC communications protocols support the negotiation of transfer syntax. However, at present, the outcome of a transfer-syntax negotiation is always NDR.)

- RPC protocol version numbers

  The client and server runtimes must use compatible versions of the RPC protocol specified by the client in the protocol sequence. The major version number of the RPC protocol used by the server must equal the specified major version number. The minor version number of the RPC protocol used by the server must be greater than or equal to the specified minor version number.

Server Binding Information

Binding information for a server is known as server binding information. A binding handle that refers to server binding information is known as a server binding handle. The use of server binding handles differs on servers and clients.

Server Binding On a Server

Servers use a list of server binding handles. Each represents one way to establish a binding with the server. Before exporting binding information to a namespace, a server tells the RPC runtime which RPC protocol sequences to use for the RPC interfaces the server supports. For each protocol sequence, the server runtime creates one or more server binding handles. Each server binding handle refers to binding information for a single potential binding, including a protocol sequence, a network (host) address, an endpoint (server address), a transfer syntax, and an RPC protocol version number.

Server Binding On a Client

A client uses a single server binding handle that refers to the server binding information the client needs for making one or more remote procedure calls to a given server. Server binding information on a client contains binding information for one potential binding.

On a client, server binding information always includes a protocol sequence and the network address of the server's host system. However, sometimes a client obtains binding information that lacks an endpoint, resulting in a partially bound binding handle. A partially bound binding handle corresponds to a system, but not to a particular server instance. When a client makes a remote procedure call using a partially bound binding handle, the client runtime gets an endpoint either from the interface specification (if one a well-known endpoint is specified) or from the endpoint map on the server's system. Bindings almost never use well-known endpoints. Adding the endpoint to the server binding information results in a fully bound binding handle, which contains an endpoint and corresponds to a specific server instance. Note clients can get a partially bound handle even if a server is not running.

Defining a Compatible Server

Compatible binding information identifies a server whose communications capabilities (RPC protocol and protocol version, network and transport protocols, and transfer syntax) are compatible with those of the client. Compatible binding information is sufficient for establishing a binding. However, binding information is insufficient for ensuring that the binding is to a compatible server.

The additional information required that a client imposes on the RPC runtime includes an RPC interface identifier and an object UUID, as follows:

- Interface identifier

  The interface UUID and version numbers of an RPC interface:

  – Interface UUID: The interface UUID, unlike the interface name, clearly identifies the RPC interface across time and space.

  – Interface version number: The combined major and minor version numbers identify one generation of an interface.

    Version numbers allow multiple versions of an RPC interface to coexist. Strict rules govern valid changes to an interface and determine whether different versions of an interface are compatible. For a description of these rules, see “Chapter 18. Interface Definition Language” on page 357 on IDL syntax and usage.

    The runtime uses the version number of an RPC interface to decide whether the version offered by a given server is compatible with the version requested by a client. The offered and requested interface are compatible under the following conditions:

    - The interface requested by the client and the interface offered by the server have the same major version number.
    - The interface requested by the client has a minor version number less than or equal to that of the interface offered by the server.

- Object UUID

  A UUID that identifies a particular object.

  An object is a distinct computing resource, such as a particular database, a specific RPC service that a remote procedure can access, and so on; for example, personal calendars may be RPC objects to a calendar service. Accessing an object requires including its object UUID with the binding information used for establishing a binding. A client can request a specific RPC object when requesting new binding information, or the client can ask the runtime to associate an object UUID with binding information the client already has available.

  Sometimes the object UUID is the nil UUID; when calling an RPC runtime routine, you can represent the nil UUID by specifying NULL. In this case, the object UUID does not represent any object. Often, however, the object UUID represents a specific RPC object and is a non-nil value. To create a non-nil object UUID, a server calls the **uuid_create()** routine, which returns a UUID that the server then associates with a particular object.

  If a client requests a non-nil object UUID, the client runtime uses that UUID as one of the criteria for a compatible server. When searching the namespace for server binding information, the client runtime looks for the requested interface identifier and object UUID. The endpoint map service uses this same information to help find a compatible server.

Figure 23 illustrates the aspects of a server and its system that are identified by the client's server binding information, requested interface identifier, and requested object UUID.

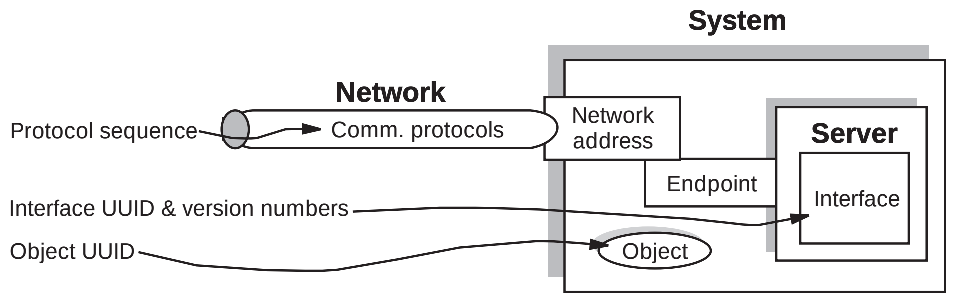


Figure 23. Information Used to Identify a Compatible Server

How Clients Obtain Server Binding Information

When a client initiates a series of related remote procedure calls, the RPC runtime tries to establish a binding, which requires the address of a compatible server. An RPC client can use compatible binding information obtained from either a namespace or from a string representation of the binding information. Using the namespace is the most common approach.

Establishing a binding also involves requesting an endpoint from the endpoint mapper of the server's system.

Binding Information in a Namespace

Usually, a server exports binding information for one or more of its interface identifiers and its object UUIDs, if any, to an entry in a namespace. The namespace is provided by a directory service such as the DCE Cell Directory Service (CDS). The namespace entry to which a server exports binding information is known as a server entry.

To learn about a server that offers a given RPC interface and object, if any, a client can import binding information from a server entry belonging to that server. A client can delegate the finding of servers from the namespace to a stub. In this case, if a binding is accidentally broken, the RPC runtime automatically tries to establish a new binding with a compatible server.

Advantages of using a directory service to obtain binding information include the following:

- It is convenient for large RPC environments. Initial overhead of understanding and configuring a directory service is balanced by easier management over time.
- Management of data in a directory service is more automated.
- It is effective in dynamic end-user environments.
- Binding information is stored in a named server entry. Data can be dynamic. Servers can automatically place their binding information in the namespace.

Changes in binding information are made once by a server or administrator and then propagated automatically by the directory service to the replicas of the data.
- There is centralized administration of data in a namespace. Sophisticated access control is possible.
- It supports searching for and choosing services based on an interface identifier and object UUID. Clients access data by specifying an entry name. Groups and profiles in directory service entries provide search paths for importing binding information.

Binding Information in Strings

Occasionally, a client can receive binding information in the form of a string (also known as a string binding). The client can receive a string binding (or the information to compose a string binding) from many sources; for example, an application-specific environment variable, a file, or the application user. The client must call the RPC runtime to convert a string binding to a binding handle. The runtime returns this binding handle to the client to use for remote procedure calls.

String representations of binding information have several possible components. The binding information includes an RPC protocol sequence, a network address, and an endpoint. The protocol sequence is mandatory; the endpoint is optional; and for a server on the client's system, the network address is optional. Also, a string binding optionally associates an object UUID with the binding information.

The string bindings have the following format:

obj-uuid@*rpc-protocol-seq*:*network-addr*[*endpoint*,*option-name*=*opt-value*...]

or

obj-uuid@*rpc-protocol-seq*:*network-addr*[**endpoint**=*endpoint*,*option-name*=*opt-value*...]

The following example string binding contains all possible components:

```
b07122e2-83df-11c9-be29-08002b1110fa@ncacn_ip_tcp:130.105.1.1.123[2001]
```

The following example string binding contains only the protocol sequence and network address:

```
ncacn_ip_tcp:130.105.1.1.123
```

For more information about the string binding format, see the RPC introduction reference page, **rpc_intro(3rpc)**.

String bindings are useful in small environments; for example, when developing and testing an application. However, string bindings are inappropriate as the principal way of providing binding information to clients. Applications should use the directory service to advertise binding information.

Client Binding Information for Servers

When making a remote procedure call, the client runtime provides information about the client to the server runtime. This information, known as client binding information, includes the following information:
- The address where the call originated (network address and endpoint)
- The RPC protocol used by the client for the call
- The object UUID that a client requests

- The client authentication information (if present)

The server runtime maintains the client binding information and makes it available to the server application by a client binding handle. Figure 24 illustrates the relationships between what a client supplies when establishing a binding and the corresponding client binding information.

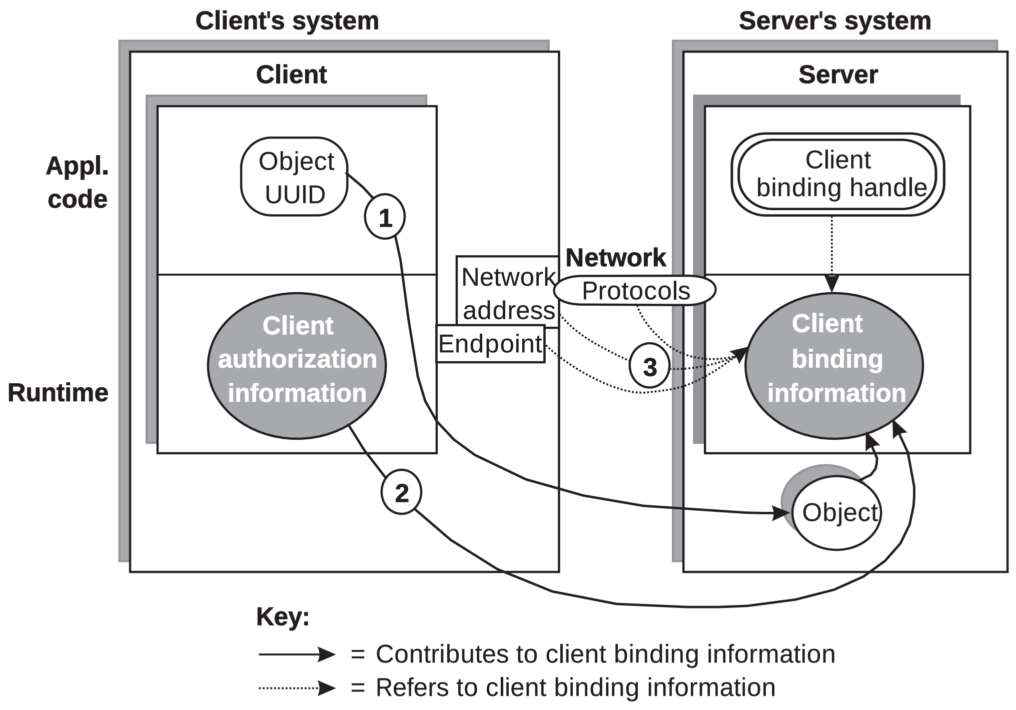


Figure 24. Client Binding Information Resulting from a Remote Procedure Call

The callouts in the figure refer to the following:

1. The requested object UUID, which may be the nil UUID
2. Client authentication information, which is optional
3. The address from which the client is making the remote procedure call, which the communications protocols supply to the server

A server application can use the client binding handle to ask the RPC runtime about the object UUID requested by a client or about the client’s authentication information.

Endpoints

An endpoint is the address of a specific server instance on a host system. Two kinds of endpoints exist: well-known endpoints and dynamic endpoints.

Well-Known Endpoints

A well-known endpoint is a preassigned stable address that a server uses every time it runs. Well-known endpoints typically are assigned by a central authority responsible for a transport protocol; for example, the Internet Assigned Numbers

Authority assigns endpoint values for the IP family of protocols. If you use well-known endpoints for a server, you should register them with the appropriate authority.

Well-known endpoints can be declared for an interface (in the interface declaration) or for a server instance, as follows:

- Any interface definition can be associated with one or more endpoints, along with the RPC protocol sequence corresponding to each endpoint (the **endpoint** attribute).

  When compiling an interface, the IDL compiler stores each combination of endpoint and protocol sequence in the interface specification. If a call is made using binding information that lacks an endpoint, the RPC runtime automatically looks in the interface specification for a well-known endpoint specified for the protocol sequence obtained from the binding information. If the interface specification contains an appropriate endpoint, the runtime adds it to the binding information.
- Alternatively, server-specific, well-known endpoints can be defined in server application code. When asking the runtime to use a given protocol sequence, the server supplies the corresponding endpoints to the RPC runtime. On a given system, each endpoint can be used by only one server at a time. If server application code contains a hardcoded endpoint or the server's installers always specify the same well-known endpoint, only one instance of the server can run per system.

When a server exports its binding information to a namespace server entry, the export operation includes any well-known endpoints within the server binding information stored in the server entry.

Dynamic Endpoints

A dynamic endpoint is requested and assigned at runtime. For some transport protocols, the number of endpoints is limited; for example, TCP/IP and UDP/IP use a 16-bit number for endpoints, which allows 65,535 endpoints. When the supply of endpoints for a transport protocol is limited, the protocol ensures an adequate supply of endpoints by limiting the portion that can be reserved as well-known endpoints. A transport, on request, dynamically makes its remaining endpoints available on a first-come, first-served basis to specific processes such as RPC server instances.

When a server requests dynamic endpoints, the server's RPC runtime asks the operating system for a unique dynamic endpoint for each protocol sequence the server is using. For a given protocol sequence, the local implementation of the corresponding transport protocol provides the requested endpoints. When an RPC server with dynamic endpoints stops listening, its dynamic endpoints are released.

Because of the transient nature of dynamic endpoints, the NSI of the RPC API does not export them to a namespace; however, NSI does export the rest of the server's binding information. References to expired endpoints would remain indefinitely in server entries, causing clients to import and try, unsuccessfully, to establish bindings to nonexistent endpoints. Therefore, the export operation removes dynamic endpoints before adding binding information to a server entry; the exported server address contains only network addressing information. The import operation returns a partially bound binding handle. The client makes its first remote procedure call with the partially bound handle, and the endpoint mapper service on the server's system resolves the binding handle with the endpoint of a compatible server. To make dynamic endpoints available to clients that are using partially bound binding handles, a server must register its dynamic endpoints in the local endpoint map.

By using object UUIDs, a server can ensure that a client that imports a partially bound handle obtains one of a particular server’s endpoints. This requires that the server do the following:

1. Specify a list of one or more object UUIDs that are unique to the server.
2. Export the list of object UUIDs.
3. Supply the list of object UUIDs to the endpoint map service when registering endpoints.
4. If the server provides different managers that implement an interface for different types of objects, the server must specify the type of each object.

To request binding information for a particular server, a client specifies one of the server’s object UUIDs, which is then associated with the server binding information the client uses for making a remote procedure call.

Note: If a client requests the nil object UUID when importing from a server entry containing object UUIDs, the client runtime selects one of those object UUIDs and associates it with the imported server binding information. This object UUID guarantees that the call goes to the server that exported the binding information and object UUID to the server entry.

Execution Semantics

Execution semantics identify the ability of a procedure to execute more than once during a given remote procedure call. The communications environment that underlies remote procedure calls affects their reliability. A communications link can break for a variety of reasons such as a server termination, a remote system crash, a network failure, and so forth; all invocations of remote procedures risk disruption due to communications failures. However, some procedures are more sensitive to such failures, and their impact depends partly on how reinvoking an operation affects its results.

To maximize valid outcomes for its operations, the operation declarations of an RPC interface definition indicate the effect of multiple invocations on the outcome of the operations.

Table 9 summarizes the execution semantics for DCE RPC calls.

Table 9. Execution Semantics for DCE RPC Calls

Semantics	**Meaning**	
at-most-once	The operation must execute either once, partially, or not at all; for example, adding or deleting an appointment from a calendar can use **at-most-once** semantics. This is the default execution semantics for remote procedure calls.	
idempotent	The operation can execute more than once; executing more than once using the same input arguments produces identical outcomes without undesirable side effects; for example, an operation that reads a block of an immutable file is **idempotent**. DCE RPC supports **maybe** semantics and **broadcast** semantics as special forms of **idempotent** operations.	
	Semantics	**Meaning**

Table 9. Execution Semantics for DCE RPC Calls (continued)

Semantics	Meaning	
	maybe	The caller neither requires nor receives any response or fault indication for an operation, even though there is no guarantee that the operation completed. An operation with **maybe** semantics is implicitly idempotent and must lack output parameters.
	broadcast	The operation is always broadcast to one server on each host system on the local network, rather than delivered to a specific server, and one reply is returned to the client. An operation with **broadcast** semantics is implicitly **idempotent**.

The broadcast capabilities of RPC runtime have a number of distinct limitations:

- Not all systems and networks support broadcasting. In particular, broadcasting is not supported by the RPC connection-oriented protocol.
- Broadcasts are limited to hosts on the local network.
- Broadcasts make inefficient use of network bandwidth and processor cycles.
- The RPC runtime library does not support **at-most-once** semantics for broadcast operations; it applies **idempotent** semantics to all such operations.
- The input arguments for broadcast calls are limited to 944 bytes.

Communications Failures

If a server detects a communications failure during a remote procedure call, the server runtime attempts to terminate the now orphaned call by sending a cancel to the called procedure. A *cancel* is a mechanism by which a client thread of execution notifies a server thread of execution (the to be canceled thread) to terminate as soon as possible. A cancel sent by the RPC runtime after a communications failure initiates orderly termination for a remote procedure call. (For a brief discussion of how cancels work with remote procedure calls, see the discussions with respect to Threads.)

Applications that use context handles to establish a client context require a context rundown procedure to enable the server to clean up client context when it is no longer needed. The name of the context rundown procedure is determined from the type name of the context handle declared in the interface definition; this ensures that the stub knows about the procedure in the server application code. If a communications link with a client is lost while a server is maintaining context for the client, the RPC runtime will inform the server to invoke the context rundown procedure. For a more thorough discussion of context handles see “Chapter 17. Topics in RPC Application Development” on page 313.

Scaling Applications

Unlike local applications, RPC applications require network resources, which are possible bottlenecks to scaling an RPC application. RPC clients and servers require network resources that are not required by local programs. The main network resources to consider are network bandwidth, endpoints, network descriptors (the identifiers of potential network channels such as UNIX sockets), kernel buffers and, for a connection-oriented transport, the connections. Also, RPC applications place extra demands on system resources such as memory buffers, various quotas, and the CPU.

The number of remote procedure calls that a server can support depends on various factors, such as the following:

- The resources of the server and the network
- The requirements of each call
- The number of calls that can be concurrently offered at some level of service
- The performance requirements

An accurate analysis of the requirements of a given server involves detailed work load and resource characterization and modeling techniques. Although measurement of live configurations under load will offer the best information, general guidelines apply. You should consider the connection, buffering, bandwidth, and CPU resources as the most likely RPC bottlenecks to scaling. Use these application requirements to scale resources.

Many system implementations limit the number of network connections per process. This limit provides an upper bound on the number of clients that can be served concurrently using the connection-oriented protocol. Some UNIX based systems set this limit at 64. However, except for applications that use context handles, the connection-oriented RPC runtime allows pooling of connections. Pooling permits simultaneously supporting more clients than the maximum number of connections, provided they do not all make calls at the same instant and occasionally can wait briefly.

RPC Objects

DCE RPC enables clients to find servers that offer specific RPC objects. An RPC object is an entity that an RPC server defines and identifies to its clients. Frequently, an RPC object is a distinct computing resource such as a particular database, directory, device, process, or processor. Identifying a resource as an RPC object enables an application to ensure that clients can use an RPC interface to operate on that resource. An RPC object can also be an abstraction that is meaningful to an application such as a service or the location of a server.

RPC objects are defined by application code. The RPC runtime provides substantial flexibility to applications about whether, when, and how they use RPC objects. RPC applications generally use RPC objects to enable clients to find and access a specific server. When servers are completely interchangeable, using RPC objects may be unnecessary. However, when clients need to distinguish between two servers that offer the same RPC interface, RPC objects are essential. If the servers offer distinct computing resources, each server can identify itself by treating its resources as RPC objects. Alternatively, each server can establish itself as an RPC object that is distinct from other instances of the same server.

RPC objects also enable a single server to distinguish among alternative implementations of an RPC interface, as long as each implementation operates on a distinct type of object. To offer multiple implementations of an RPC interface, a server must identify RPC objects, classify them into types, and associate each type with a specific implementation.

The set of remote procedures that implements an RPC interface for a given type of object is known as a manager. The tasks performed by a manager depend on the type of object on which the manager operates. For example, one manager of a queue-management interface may operate on print queues, while another manager may operate on batch queues.

Chapter 13. Basic RPC Routine Usage

This chapter introduces a number of basic DCE RPC routines for directory service, communications, and authentication operations and discusses major usage issues important for developing DCE RPC applications.

This chapter discusses the following topics:

- Overview of basic runtime routines
- Server initialization tasks
- How clients find servers

Overview of the RPC Routines

This section summarizes the major concerns of RPC communications, name service interface (NSI) usage, and authenticated RPCs.

Basic Operations of RPC Communications

The DCE RPC runtime provides the following communications operations for RPC applications:

- Managing communications for RPC applications

  As part of server initialization, a server sets up its communications capabilities by a series of calls to the RPC runtime. These runtime calls register the server's RPC interfaces, tell the RPC runtime what combination of communications protocols to use for the server, and register the endpoints of the server for each of its interfaces. After completing these and any other initialization tasks, the server tells the runtime to begin listening for incoming calls.
- Managing binding information

  A variety of communications operations allow servers to access and manipulate binding information. In addition, a set of communications operations enables applications to manipulate string representations of binding information (string bindings).

Basic Operations of the NSI

The NSI routines perform operations on a namespace for RPC applications. The fundamental operations include the following:

- Creating and deleting entries in namespaces
- Exporting

  A server uses the NSI export operation to place binding information associated with its RPC interfaces and objects into the namespace used by the RPC application.
- Importing

  Clients can search for exported binding information associated with an interface and object by using the NSI import operation or lookup operation. These two operations are collectively referred to as the NSI search operations.
- Unexporting

  The unexport operation enables a server to remove some or all of its binding information from a server entry.
- Managing information in a namespace

Applications use the NSI interface to place information about server entries into a namespace and to inquire about and manage that information.

Basic Operations of Authenticated RPCs

The authenticated RPC routines provide a mechanism for establishing secure communications between clients and servers.

To engage in authenticated RPC, a client and server must agree on the authentication service to be used. The server's responsibility is to register its principal name and the authentication service to be supported with the RPC runtime. The client's responsibility is to establish the authentication service, a given protection level, and an authorization service for the server binding handle. The protection level determines the degree of protection applied to individual messages between the client and server. The authorization service determines the form in which the client's credentials will be presented to the server (for access checking).

Once authenticated RPC has been established between a client and server, the client issues remote procedure calls in the usual fashion, with all authentication and protection being handled by the DCE Security Service and the RPC runtime.

Table 10 relates several of the RPC runtime operations just discussed with specific routines or sets of routines.

Table 10. Basic Runtime Routines

Description of Operation	Usage	Routine Name(s)
Communications Routines		
Set the type of an RPC object with the RPC runtime	Server	**rpc_object_set_type()**
Register RPC interfaces	Server	**rpc_server_register_if()**
Select RPC protocol sequences	Server	**rpc_network_inq_protseqs()**, **rpc_server_use_*protseq*_...()**
Obtain server binding handles	Server	**rpc_server_inq_bindings()**
Register endpoints	Server	**rpc_ep_register()**, **rpc_ep_register_no_replace()**
Unregister endpoints	Server	**rpc_ep_unregister()**
Listen for calls	Server	**rpc_server_listen()**
Manipulate string representations of binding information (string bindings)	Client	**rpc_binding_from_string_binding()**
	Client, Server	**rpc_binding_to_string_binding()**, **rpc_string_binding_compose()**, **rpc_string_binding_parse()**
Change the RPC object in server binding information	Client	**rpc_binding_set_object()**
Convert a client binding handle to a server binding handle	Server	**rpc_binding_server_from_client()**
Name Service Interface Routines		
Export binding information to a namespace	Server	**rpc_ns_binding_export()**

Table 10. Basic Runtime Routines (continued)

Description of Operation	Usage	Routine Name(s)
Search a namespace for binding information	Client	**rpc_ns_binding_import_...()**, **rpc_ns_binding_lookup_...()**, **rpc_ns_binding_select()**
Authentication Routines		
Authentication and authorization	Server, Client	**rpc_*auth...()**

Server Initialization Using the RPC Routines

Before a server can receive any remote procedure calls, it should usually initialize itself by calling the **dce_server_register()** routine so that the server is properly recognized by DCE. However, servers can instead use a series of the lower-level RPC runtime routines. The server initialization code, written by the application developer, varies among servers. However, every server must set up its communications capabilities, which usually involves most of the following tasks:

1. Assigning types to objects
2. Registering at least one interface
3. Specifying which protocol sequences the server will use
4. Obtaining a list of references to a server's binding information (a list of binding handles)
5. Registering endpoints
6. Exporting binding information to a server entry or entries in the namespace
7. Listening for remote procedure calls
8. Performing cleanup tasks including unregistering endpoints

The following pseudocode illustrates the calls a server makes to accomplish these basic initialization tasks:

```
/* Initialization tasks */

  rpc_object_set_type(...);
  rpc_server_register_if(...);
  rpc_server_use_all_protseqs(...);
  rpc_server_inq_bindings(...);
  rpc_ep_register(...);
  rpc_ns_binding_export(...);
  rpc_server_listen(...);

/* Cleanup tasks */

  rpc_ep_unregister(...);
```

Assigning Types to Objects

An object type is a mechanism for associating a set of RPC objects and the manager whose remote procedures implement an RPC interface for those objects. Object types allow an application to cluster objects, such as computing resources, according to any relevant criteria. For example, a single accounting interface can be implemented to operate on accounting databases that contain equivalent information but that are formatted differently; each database format represents a distinct type.

To simultaneously offer alternative implementations of an RPC interface for different types of objects, a server uses alternative managers. Servers that implement each of their interfaces with only one manager can usually avoid the tasks associated with assigning object types. However, when a server offers multiple managers, each manager must be dedicated to operating on a separate type of object. In this case, a server must classify some or all of its objects into types; for example, a calendar application that specifies one non-nil type UUID for departmental calendars and another non-nil type UUID for personal calendars.

By default, objects have the nil type. Only a server that implements different managers for different objects or sets of objects needs to type classify its RPC objects. To type classify an object, a server associates the object UUID of the object with a single type UUID by calling the **rpc_object_set_type()** procedure separately for each object. To create a UUID, a server calls the **uuid_create()** routine.

The exact implementation of a manager can vary with the type of object on which each manager operates. For example, a queue-management interface may be implemented to manage print queues as objects in one case and to manage batch queues as objects in another. Figure 25 illustrates the use of type UUIDs to identify two types of managers.

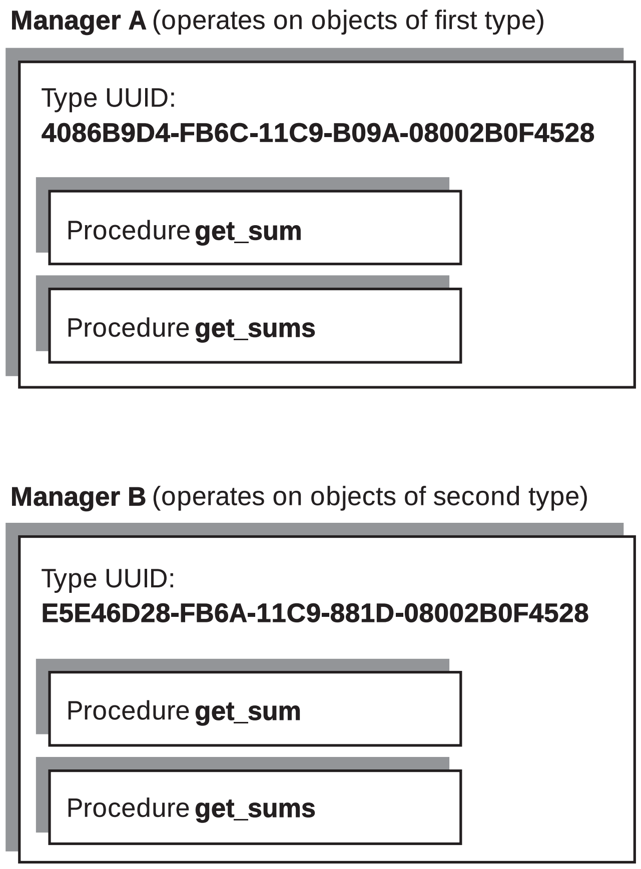

Manager A (operates on objects of first type)
Type UUID:
4086B9D4-FB6C-11C9-B09A-08002B0F4528
Procedure get_sum
Procedure get_sums
Manager B (operates on objects of second type)
Type UUID:
E5E46D28-FB6A-11C9-881D-08002B0F4528
Procedure get_sum
Procedure get_sums

Figure 25. Manager Types

When the server receives an incoming call that specifies an object UUID, the server dispatches the call to the manager for the type to which the object belongs. For information on how a type is used to select a manager for an incoming call, see "Chapter 17. Topics in RPC Application Development" on page 313.

Registering Interfaces

A server calls the **rpc_server_register_if()** routine to tell the RPC runtime about a specific RPC interface. Registering an interface informs the runtime that the server is offering that interface and makes it available to clients. A server can register any number of interfaces with the RPC runtime by calling the **rpc_server_register_if()** routine once for each set of procedures, or manager, that implements an interface.

To offer more than one manager for an interface, a server must register each manager separately.

When registering an interface, the server provides the following information:

- Interface specification

  This is a reference to information about an RPC interface as offered by its server stub. The DCE IDL compiler generates an interface specification as part of the stub code. For a specific version of an interface, all managers use the same interface specification. Information in an interface specification that concerns application developers includes the following:

  - The interface identifier (UUID and major and minor version numbers)
  - The supported transfer syntaxes
  - A list of any well-known endpoints (and their associated protocol sequences) specified in the interface definition (**.idl**) file
  - The interface's default manager entry point vector (manager EPV), if present

    A default manager EPV, constructed using the operation names of the interface definition, is typically generated for stubs by the DCE IDL compiler (the **--no_mepv** compiler option suppresses this feature).
- A type UUID for the manager

  Each implementation of an interface, a manager, is represented by a type UUID.
- A manager EPV for the interface

  A server can register a given interface more than once by specifying a different type UUID and manager EPV each time it calls **rpc_server_register_if()**.

  A manager EPV is a list of the addresses (the entry points of the remote procedures provided by the manager) that represent the location of each remote procedure implementation. A manager EPV must contain exactly one entry point for each procedure defined in the interface definition.

  The server can use the default manager EPV only once, and only for a manager that uses the procedure names as they are declared in the interface definition. For any additional manager of the RPC interface, (and if the server needs to rename the implemented procedures), the server must create and register a unique manager EPV. Also, each manager must be associated with a distinct type UUID.

Selecting RPC Protocol Sequences

A server can inquire about whether the local RPC runtime supports a specific protocol sequence by using the **rpc_network_is_protseq_valid()** routine. The server can also use the **rpc_network_inq_protseqs()** routine to ask the RPC runtime for a list of all protocol sequences supported by both the RPC runtime and the operating system.

To prepare to receive remote procedure calls, a server uses **rpc_server_use_all_protseqs()** or **rpc_server_use_protseq()** calls to tell the RPC runtime to use at least one protocol sequence. For each protocol combination, the RPC runtime creates one or more binding handles with dynamic endpoints on which the server will listen for remote procedure calls. The server then can use a list of these binding handles to register dynamic endpoints in the endpoint map and to export its binding information (except the endpoints) to the name service.

As an option, an interface can contain one or more well-known endpoints, each of which is accompanied by a protocol sequence. A server uses the **rpc_server_use_all_protseqs_if()**, **rpc_server_use_protseq_if()**, or **rpc_server_use_protseq_ep()**, to notify the RPC runtime about which protocol sequence and well-known endpoint combinations will be used.

A server can use any protocol sequence declared in an interface endpoint declaration, or the server can ignore the endpoint declarations, as long as it registers at least one endpoint.

Obtaining a List of Server Binding Handles

After a server passes to the RPC runtime the protocol sequences over which it will listen for remote procedure calls, the RPC runtime constructs server binding handles. Each binding handle refers to a complement of binding information that defines one potential binding; that is, a specific 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.

Before registering endpoints or exporting binding information, a server must obtain a list of its binding handles from the RPC runtime by using the **rpc_server_inq_bindings()** routine. The server passes this list back to the runtime as an argument when registering endpoints and exporting binding information.

Registering Endpoints

Servers can use well-known or dynamic endpoints with any protocol sequence.

When a server asks the runtime to use a dynamic endpoint with a protocol sequence, the runtime asks the operating system to generate the endpoint. To use the dynamic endpoints, a server must register the server’s binding information, including the endpoints, by using the **rpc_ep_register()** routine. For each combination of RPC interface identifier, object UUID, and binding information that the server offers, the endpoint mapper service creates an element in the local endpoint map.

A server does not necessarily need to register well-known endpoints; however, by registering well-known endpoints, the server ensures that clients can always obtain them. Registration also makes the endpoints accessible to administrators, who can use the DCE control program, **dcecp**, to show the map elements of an endpoint map by using the **endpoint show** operation.

Servers can remove map elements from a local endpoint map by using the **rpc_ep_unregister()** routine. Servers should unregister endpoints after they stop listening.

Making Binding Information Accessible to Clients

A server needs to make its binding information accessible to clients. Usually, a server uses the NSI export operation to place its binding information into a server entry. However, it is also possible for servers to make string bindings accessible to clients. In any case, the server obtains its binding information from the runtime by first using the **rpc_server_inq_bindings()** routine to ask for a list of binding handles.

Using String Bindings to Provide Binding Information

While implementing and debugging a server program you may temporarily want to communicate binding information to clients by using string bindings. This allows a server to establish a client/server relationship without registering endpoints in the local endpoint map or exporting binding information to a namespace.

The server can convert into a string each binding handle in the list obtained from the **rpc_server_inq_bindings()** call by calling **rpc_binding_to_string_binding()**. The resulting string binding is always fully bound. The server then makes some or all of its string bindings available to clients somehow; for example, by placing the string bindings in a file to be read by clients or users or both.

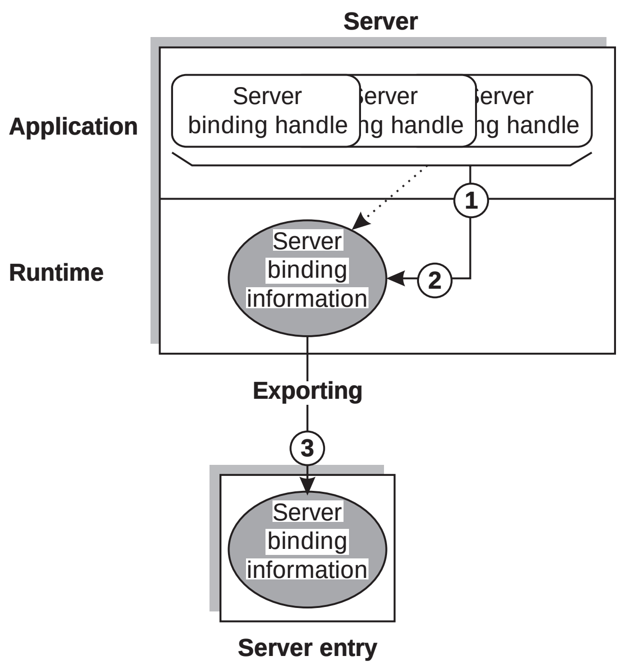
Exporting Binding Information

Servers can export binding information (and interface identifiers) or objects or both by calling the **rpc_ns_binding_export()** routine. To export binding information associated with a given RPC interface, a server uses an interface handle. The interface handle is created by the IDL compiler as a reference to information about the interface that the compiler stores in an interface specification.

To refer to binding information, the application code obtains a list of server binding handles from the RPC runtime and passes the list to the export operation. The list contains binding handles for all the protocol sequence and endpoint combinations that the server has requested; it does this by calling the use-protocol-sequence operations. However, the server can remove any of those binding handles from the list before exporting it. This enables a server to export the binding information associated with a subset of its binding handles.

To export object UUIDs, a server application must provide a list of object UUIDs for the RPC objects it offers. The server can generate these object UUIDs itself or obtain them from some application-specific source such as an object-UUID database. All object UUIDs in a given server entry are associated with every interface UUID and server address in the entry.

Figure 26 on page 190 illustrates how server binding handles in the application code refer to server binding information in the runtime, which is exported to the name service.



◄······ = Reference to binding information

Figure 26. Exporting Server Binding Information

A server entry must belong exclusively to a server running on a given host. If there are identical, interchangeable instances of a server on the host, they can share a single set of server entries. However, if clients need to distinguish between coexisting instances of a server (for example, when each offers a different RPC object), each instance requires its own server entry.

Note: CDS databases are subject to access control. To access entries in a CDS database, you need access control list (ACL) permissions. Depending on the NSI operation, you need ACL permissions to the parent directory, the CDS object entry, or both. If you need ACL permissions, see your CDS administrator.

The ACL permissions are as follows:

- To create an entry, you need **insert** permission to the parent directory.
- To read an entry, you need **read** permission to the CDS object entry.
- To write to an entry, you need **write** permission to the CDS object entry.
- To delete an entry, you need **delete** permission either to the CDS object entry or to the parent directory.
- To test an entry, you need either **test** permission or **read** permission to the CDS object entry.

Note that **write** permission does not imply **read** permission.

Listening for Calls

When a server is ready to accept remote procedure calls, it initiates listening, specifying the maximum number of calls it can execute concurrently; it does this by calling the **rpc_server_listen()** routine. If a server allows concurrent calls, its remote procedures are responsible for concurrency control. If executing a set of remote procedures concurrently requires concurrency control and a server lacks this control, the server must allow only one call at a time.

Under normal circumstances, the **rpc_server_listen()** routine does not return but the RPC runtime continues listening for new remote procedure calls to the server's registered interfaces until one of the following events occurs:

- Any of the server's procedures makes a local management call to stop a server from listening for future remote procedure calls.
- For applications whose servers enable clients to stop servers from listening, a client makes a remote management call to stop a server from listening for future remote procedure calls.

On receipt of a stop listening request, the RPC runtime stops accepting new remote procedure calls for all registered interfaces. However, currently executing calls are allowed to complete. After all executing calls complete, the listen operation stops listening and returns control to the server. Servers should unregister endpoints after they stop listening.

How Clients Find Servers

A client runtime can obtain server binding information from a namespace. Alternatively, a client can obtain server binding information in string format from an application-specific source such as a file. Runtime routines enable client applications to obtain server binding handles that refer to server binding information obtained from either source.

Searching a Namespace

To obtain binding information from a namespace, a client can do one of the following:

- The client must call the import routines **rpc_ns_binding_import_begin()**, **rpc_ns_binding_import_next()**, and **rpc_ns_binding_import_done()** to obtain a binding handle for a compatible server.
- The client must call the lookup routines **rpc_ns_binding_lookup_begin()**, **rpc_ns_binding_lookup_next()**, and **rpc_ns_binding_lookup_done()** to obtain a list of binding handles for a compatible server. Select a binding handle from the list by calling either of the following:
  - The NSI select routine **rpc_ns_binding_select()**, which selects a binding handle at random
  - A user-defined select routine, which implements an application-specific selection algorithm
- The client must use the automatic method of binding management to make the client stub transparently manage binding information.

  In this case, the application code lacks any calls to the NSI interface. However, the automatic method does require the client to identify the directory service entry at which to begin the search for binding information. The client must specify the starting entry name as the value of the NSI-defined **RPC_DEFAULT_ENTRY** environment variable.

An NSI import or lookup operation searches server entries for a compatible server. On finding such a server entry, the search operation copies the server binding information associated with the requested interface and an object UUID. The search operation then creates a randomly ordered list of server binding handles to refer to the potential bindings represented by the binding information.

Figure 27 illustrates the use of a server binding handle to refer to server binding information selected by an import operation.

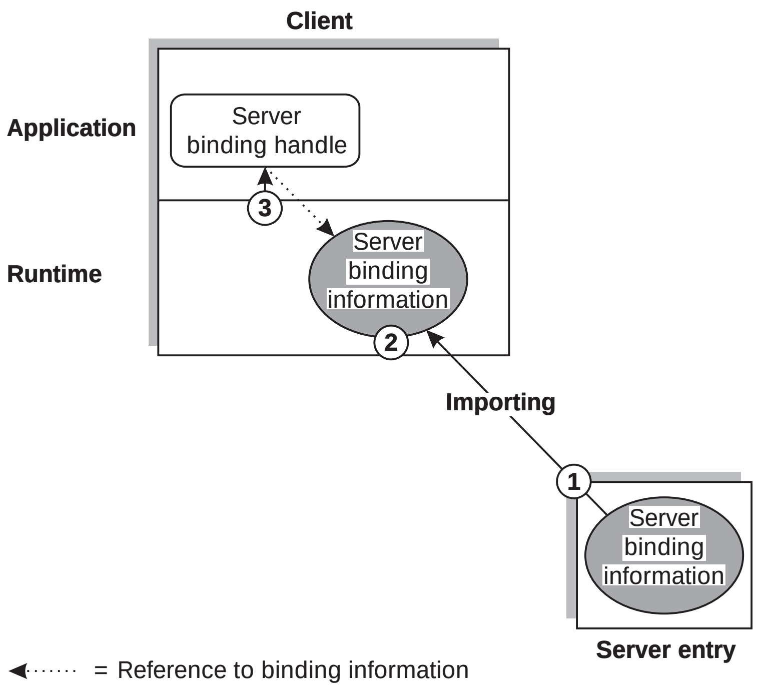

Client
Application
Server binding handle
3
Runtime
Server binding information
2
Importing
1
Server binding information
Server entry
= Reference to binding information

Figure 27. Importing Server Binding Information

The callouts in the figure refer to the following operations:

1. The import operation looks up binding information of a server that is compatible with the client.

   The import operation finds a server entry based on the specified interface identifier, and then looks at the list of object UUIDs. If the importing client specifies a non-nil object UUID, the import operation looks for and returns that object UUID. If the client specifies the nil object UUID and the server entry contains any object UUIDs, the import operation selects and returns one UUID at random. If the entry lacks any object UUIDs, the import operation returns the nil UUID.
2. The import operation fetches the compatible binding information and creates a binding handle for each potential binding represented in the binding information.
3. The import operation then selects a binding handle at random and passes it to the client application.

Using String Bindings to Obtain Binding Information

To use a string binding, a client starts with either an existing string binding or with the components of the binding information. Do *not* hardcode string bindings into application code. Rather, specify them at runtime using a command argument, environment variable, file, or other means. The simplest way to specify a string binding is for a user to supply a string binding manually to a client. However, this manual approach is awkward for users who must know how to obtain and manipulate the string bindings. Also, if binding information changes, the users are responsible for updating any string bindings used by their clients. Reducing manual intervention in the use of string bindings requires that an application provide its own mechanisms for storing, maintaining, and accessing binding information. In contrast, a directory service such as CDS provides these mechanisms automatically to applications that store binding information in a namespace.

Regardless of how a client obtains a string binding, before establishing a binding, the client must ask the RPC runtime for a binding handle that refers to the server binding information depicted in the string binding. The client converts the string binding into a server binding handle by calling the **rpc_binding_from_string_binding()** routine.

The following pseudocode lists the calls for composing a string binding and for using it to obtain a server binding handle:

```
rpc_string_binding_compose(...);

rpc_binding_from_string_binding(...);
 .
 .
 .
rpc_string_free(...);
```

Chapter 14. RPC and Other DCE Components

This chapter discusses aspects of the internal behavior of remote procedure calls that are significant for advanced RPC programmers, including the following topics:

- Threads of execution in RPC applications
- Authenticated remote procedure calls
- Using the Name Service Interface

DCE RPC is a fully integrated part of the distributed computing environment. The communications capabilities of DCE RPC are used by clients and servers of other DCE components. In turn, RPC uses services provided by DCE Threads, the DCE Security Service, and the DCE Cell Directory Service.

A thread is a single sequential flow of control with one point of execution on a single processor at any instant. Multiple threads can coexist in a single process. DCE RPC uses threads internally for its own operations. DCE RPC also provides an environment where RPC applications can use thread services.

The DCE RPC runtime provides RPC applications with a programming interface to the security service. The RPC authentication interface enables RPC clients and servers to mutually authenticate (that is, prove the identity of) each other. An authenticated remote procedure call provides client authorization information and authentication information to servers. Authorization information includes the credentials a client has and the identities a client is associated with at the time of a call. By comparing client authorization information to access control lists, a server can find out whether a client is eligible to use a requested remote procedure. Client authentication information identifies a client to a server.

To help RPC clients find RPC servers, RPC applications typically use a namespace. A namespace is a collection of information about applications, systems, and any other relevant computing resources. A namespace is maintained by a directory service such as CDS. DCE RPC provides a Name Service Interface (NSI) that is independent of any particular directory service.

NSI communicates with supported directory services for both RPC applications and the RPC control program. NSI insulates RPC applications from the intricacies of using a directory service. An RPC server uses NSI to store information about itself in a namespace, and a client uses NSI to access information about a server that meets the client’s requirements for a specific RPC interface and object, among other things. The client uses this information to establish a relationship, known as a binding, with the server.

Threads of Execution in RPC Applications

Each remote procedure call occurs in an execution context called a thread. A thread is a single sequential flow of control with one point of execution on a single processor at any instant. A thread created and managed by application code is an application thread.

Traditional processing occurs exclusively within local application threads. Local application threads execute within the confines of one address space on a local system and pass control exclusively among local code segments, as illustrated in Figure 28 on page 196.

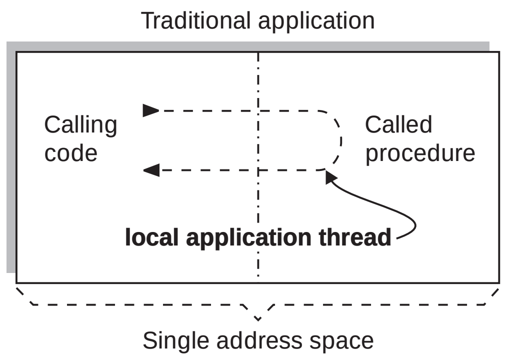


Figure 28. Local Application Thread During a Procedure Call

RPC applications also use application threads to issue both remote procedure calls and runtime calls, as follows:

- An RPC client contains one or more client application threads; that is, a thread that executes client application code that makes one or more remote procedure calls.
- A DCE RPC server uses one server application thread to execute the server application code that listens for incoming calls.

In addition, for executing called remote procedures, an RPC server uses one or more call threads that the RPC runtime provides. As part of initiating listening, the server application thread specifies the maximum number of concurrent calls it will execute. The maximum number of call threads in multithreaded applications depends on the design of the application. The RPC runtime creates the same number of call threads in the server process.

The number of call threads is significant to application code. When using only one call execution thread, application code does not have to protect itself against concurrent resource use. When using more than one call thread, application code must protect itself against concurrent resource use.

Figure 29 on page 197 shows a multithreaded server with a maximum of four concurrently executing calls. Of the four call threads for the server, only one is currently in use; the other three threads are available for executing calls.

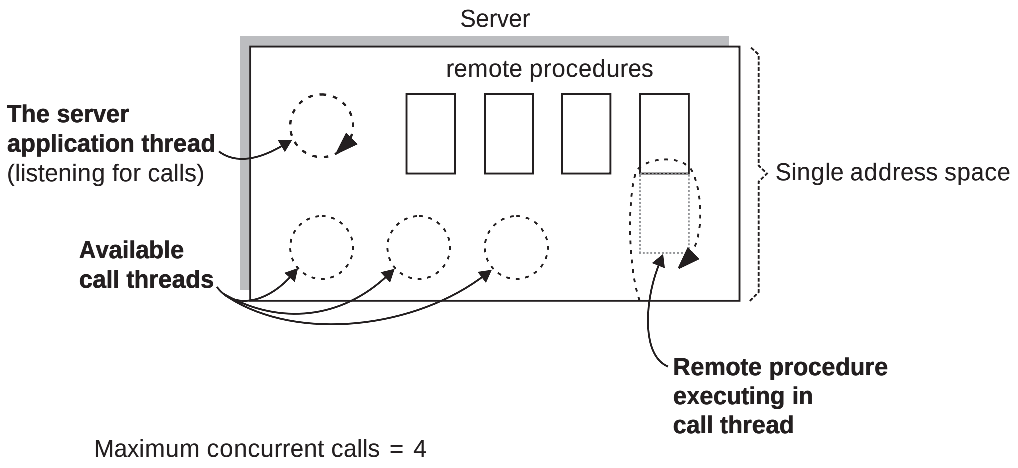

Server
remote procedures
The server application thread (listening for calls)
Single address space
Available call threads
Remote procedure executing in call thread
Maximum concurrent calls = 4

Figure 29. Server Application Thread and Multiple Call Threads

Remote Procedure Call Threads

In distributed processing, a call extends to and from client and server address spaces. Therefore, when a client application thread calls a remote procedure, it becomes part of a logical thread of execution known as an RPC thread. An RPC thread is a logical construct that encompasses the various phases of a remote procedure call as it extends across actual threads of execution and the network. After making a remote procedure call, the calling client application thread becomes part of the RPC thread. Usually, the RPC thread maintains execution control until the call returns.

The RPC thread of a successful remote procedure call moves through the execution phases illustrated in Figure 30.

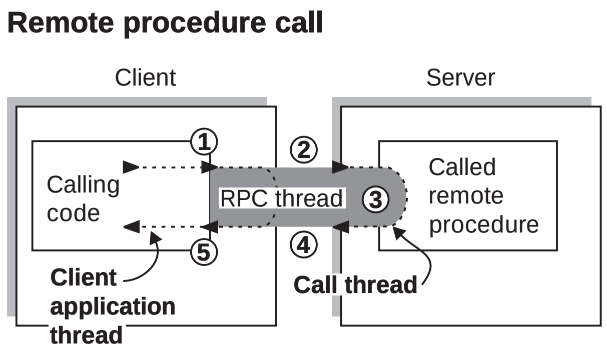


Figure 30. Execution Phases of an RPC Thread

The execution phases of an RPC thread in the preceding figure include the following operations:

1. The RPC thread begins in the client process, as a client application thread makes a remote procedure call to its stub; at this point, the client thread becomes part of the RPC thread.
2. The RPC thread extends across the network to the server address space.

3. The RPC thread extends into a call thread, where the remote procedure executes. While a called remote procedure is executing, the call thread becomes part of the RPC thread. When the call finishes executing, the call thread ceases being part of the RPC thread.
4. The RPC thread then retracts across the network to the client.
5. When the RPC thread arrives at the calling client application thread, the remote procedure call returns any call results and the client application thread ceases to be part of the RPC thread.

Figure 31 shows a server executing remote procedures in its two call threads, while the server application thread listens.

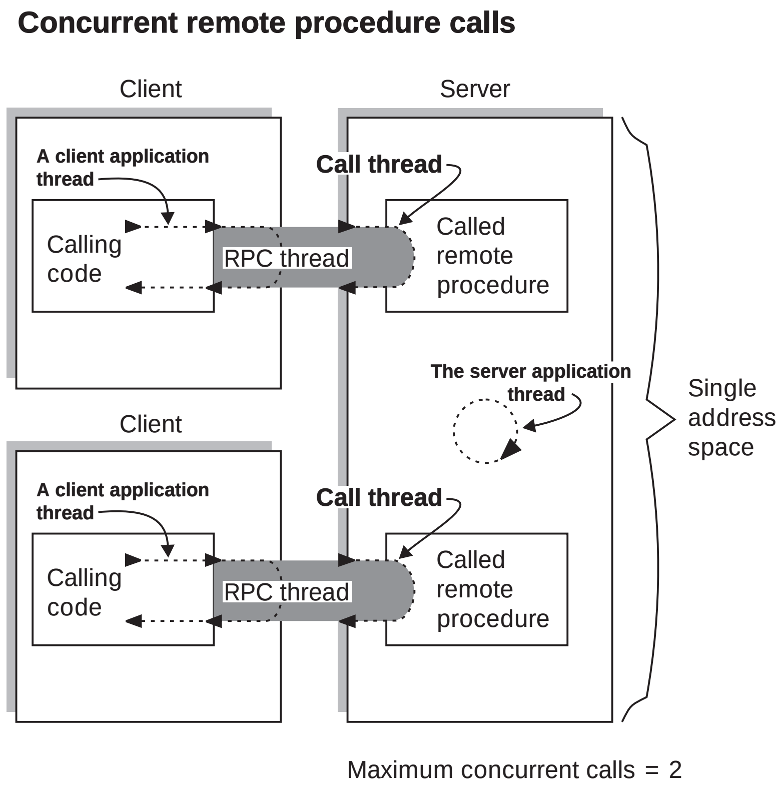


Figure 31. Concurrent Call Threads Executing in Shared Address Space

Note: Although a remote procedure can be viewed logically as executing within the exclusive control of an RPC thread, some parallel activity does occur in both the client and server.

An RPC server can concurrently execute as many remote procedure calls as it has call threads. When a server is using all of its call threads, the server application thread continues listening for incoming remote procedure calls. While waiting for a call thread to become available, DCE RPC server runtimes can queue incoming calls. Queuing incoming calls avoids remote procedure calls failing during short-term congestion. The queue capacity for incoming calls is implementation dependent; most implementations offer a small queue capacity. The queuing of incoming calls is discussed in “Chapter 17. Topics in RPC Application Development” on page 313, under the topic of the routing of incoming calls.

Cancels

DCE RPC uses and supports the synchronous cancel capability provided by POSIX threads (pthreads). A cancel is a mechanism by which a thread informs another thread (the canceled thread) to terminate as soon as possible. Cancels operate on the RPC thread exactly as they would on a local thread, except for an application-specified, cancel-timeout period. A cancel-timeout period is an optional value that limits the amount of time the canceled RPC thread has before it releases control.

During a remote procedure call, if its thread is canceled and the cancel-timeout period expires before the call returns, the calling thread regains control and the call is orphaned at the server. An orphaned call may continue to execute in the call thread. However, the call thread is no longer part of the RPC thread, and the orphaned call is unable to return results to the client.

A client application thread can cancel any other client application thread in the same process (it is possible, but unlikely, for a thread to cancel itself.) While executing as part of an RPC thread, a call thread can be canceled only by a client application thread.

A cancel goes through several phases. Figure 32 shows where each of these phases occur.

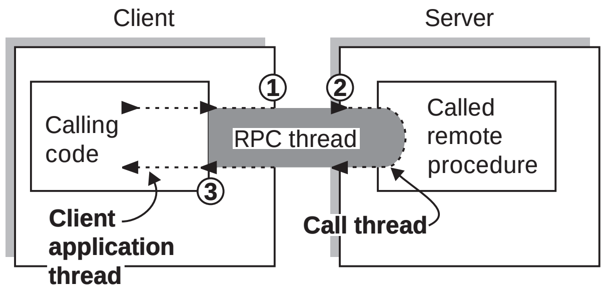


Figure 32. Phases of a Cancel in an RPC Thread

The phases of a cancel in the preceding figure include the following:

1. A cancel that becomes pending at the client application thread at the start of or during a remote procedure call becomes pending for the entire RPC thread. Thus, while still part of the RPC thread, the call thread also has this cancel pending.
2. If the call thread of an RPC thread makes a cancelable call when cancels are not deferred and a cancel is pending, the cancel exception is raised.
3. The RPC thread returns to the canceled client application thread with one of the following outcomes:
   - If a cancel exception has not been taken, the RPC thread returns normal call results (output arguments, return value, or both) with a pending cancel.
   - If the remote procedure is using an exception handler, a cancel exception can be handled. The procedure resumes, and the RPC thread returns normal call results without pending any cancel.
   - If the remote procedure failed to handle a raised cancel exception, the RPC thread returns with the cancel exception still raised. This is returned as a fault.

- If the cancel-timeout period expires, the RPC thread returns either a cancel-timeout exception or status code, depending on how the application sets up its error handling. This is true for all cases where any abnormal termination is returned.

Multithreaded RPC Applications

DCE RPC provides an environment for RPC applications that create multiple application threads (multithreaded applications). The application threads of a multithreaded application share a common address space and much of the common environment. If a multithreaded application must be thread-safe (guarantee that multiple threads can execute simultaneously and correctly), the application is responsible for its own concurrency control. Concurrency control involves programming techniques such as controlling access to code that can share a data structure or other resource to prevent conflicting overlapping access by separate threads.

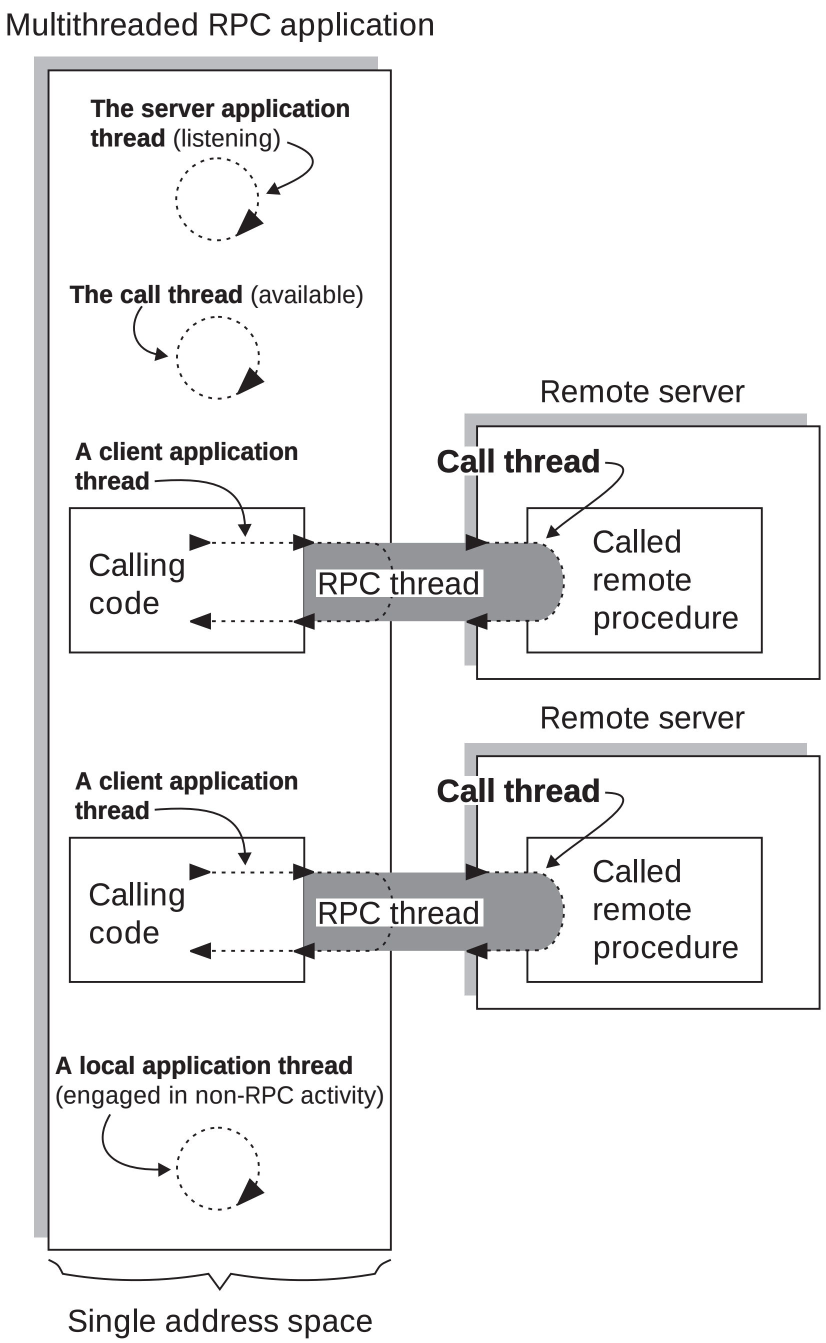
A multithreaded RPC application can have diverse activities going on simultaneously. A multithreaded client can make concurrent remote procedure calls and a multithreaded server can handle concurrent remote procedure calls. Using multiple threads allows an RPC client or server to support local application threads that continue processing independently of remote procedure calls. Also, multithreading enables the server application thread and the client application threads of an RPC application to share a single address space as a joint client/server instance. A multithreaded RPC application can also create local application threads that are uninvolved in the RPC activity of the application.

Figure 33 on page 201 shows an address space where application threads are executing concurrently.

The application threads in Figure 33 on page 201 are performing the following activities:

- The server application thread is listening for calls.
- A call thread is available to execute an incoming remote procedure call.
- One client application thread has separated from an RPC thread and another is currently part of an RPC thread.
- A local application thread is engaging in non-RPC activity.

Concurrent remote procedure calls



Multithreaded RPC application
The server application thread (listening)
The call thread (available)
A client application thread
Calling code
RPC thread
Remote server
Call thread
Called remote procedure
Remote server
A client application thread
Calling code
RPC thread
Call thread
Called remote procedure
A local application thread (engaged in non-RPC activity)
Single address space

Figure 33. A Multithreaded RPC Application Acting as Both Server and Client

Security and RPC: Using Authenticated Remote Procedure Calls

DCE RPC supports authenticated communications between clients and servers. Authenticated RPC works with the authentication and authorization services provided by the DCE Security Service.

On the application level, a server makes itself available for authenticated communications by registering its principal name and the authentication service that it supports with the RPC runtime. The server principal name is the name used to identify the server as a principal to the registry service provided by the security service. In practice, this name is usually the same as the name that the server uses to register itself with the DCE Directory Service.

A client must establish the authentication service, protection level, and authorization service that it wishes to use in its communications with a server. The client identifies the intended server by means of the principal name that the server has registered with the RPC runtime. Once the required authentication, protection, and authorization parameters have been established for the server binding handle, the client issues remote procedure calls to the server as it normally does.

The security service, in conjunction with the RPC runtime, assumes responsibility for the following:

- Authenticating the client and server in accordance with the requested authentication service
- Applying the requested level of protection to communications between the client and server
- Providing client authorization data to the server in a form determined by the requested authorization service

Note: For a detailed discussion of authentication within the context of DCE security, refer to “Chapter 24. Authentication” on page 493 of this guide.

Authentication

When a client establishes authenticated RPC, it must indicate the authentication service that it wants to use. The possible values are the following:

rpc_c_authn_none
No authentication

rpc_c_authn_dce_secret
DCE shared-secret key authentication

rpc_c_authn_dce_public
DCE public key authentication

rpc_c_authn_default
DCE default authentication service

The value **rpc_c_authn_none** is used to turn off authentication already established for a binding handle. The default authentication is DCE shared-secret (also known as private key) authentication.

Before a client and server can engage in authenticated RPC, they must “agree” on which authentication service to use. Specifically, the server must register the “agreed on” authentication service with the RPC runtime, along with the server’s principal name. For its part, the client must select the same service for the server’s binding handle. The client indicates the appropriate server by supplying the server’s principal name. If the client does not know the server’s name, it can use the **rpc_mgmt_inq_server_princ_name()** routine to determine the name.

Cross-Cell Authentication

A client can engage in authenticated RPC with a target server that is in the client's cell or in a foreign cell. In the case of cross-cell authentication, DCE security performs the necessary additional steps on behalf of the client.

To establish authenticated RPC with a foreign server, a client must supply the fully qualified principal name of the server. A fully qualified name includes the name of the cell as well as the name of the principal and takes the following form:

```
/.../ cell_name/ principal_name
```

Protection Levels

When a client establishes authenticated RPC, it can specify the level of protection to be applied to its communications with the server. The protection level determines how much of client/server messages are encrypted. As a rule, the more restrictive the protection level, the greater the impact on performance. Different levels are provided so that applications can control the protection versus performance tradeoffs.

Note that the protection level is entirely a client responsibility. When a server registers its supported authentication service with the RPC runtime, it does not specify any protection information for that service. However, the server can include the protection level used for a particular operation when deciding if the caller is authorized to perform the operation.

Authenticated RPC supports the following protection levels:

rpc_c_protect_level_default
Uses the default protection level for the specified authentication service.

rpc_c_protect_level_none
There is no protection level.

rpc_c_protect_level_connect
Performs protection only when the client establishes a relationship with the server. This level performs an encrypted handshake when the client first communicates with the server. Encryption or decryption is not performed on the data sent between the client and server. The fact that the handshake succeeds indicates that the client is active on the network.

rpc_c_protect_level_call
Performs protection only at the beginning of each remote procedure call when the server receives the request. This level attaches a verifier to each client call and server response.

This level does not apply to remote procedure calls made over a connection-based protocol sequence; that is, **ncacn_ip_tcp**. If this level is specified and the binding handle uses a connection-based protocol sequence, the routine uses the **rpc_c_protect_level_pkt** level instead.

rpc_c_protect_level_pkt
Ensures that all data received is from the expected client. This level attaches a verifier to each message.

rpc_c_protect_level_pkt_integrity
Ensures and verifies that none of the data transferred between client and server has been modified. This level computes a cryptographic checksum of

each message to verify that none of the data transferred between the client and server has been modified in transit.

This is the highest protection level that is guaranteed to be present in the RPC runtime.

rpc_c_protect_level_pkt_privacy
Performs protection as specified by all of the previous levels and also encrypts each remote procedure call argument and return values. This level encrypts all user data in each call.

This is the highest protection level, but it may not be available in the RPC runtime.

If a client wants to use the default protection level but does not know what this level is, it can use the **rpc_mgmt_inq_dflt_protect_level()** routine to determine what the default level is.

Authorization

Authorization is the process of checking a client's permissions to an object that is controlled by the server. Access checking is entirely a server responsibility and involves matching the client's credentials against the permissions associated with the object. A client's credentials consist of the principal ID and group memberships contained in the client's network login context.

Authenticated RPC supports the following options for making client authorization information available to servers for access checking:

rpc_c_authz_none
No authorization information is provided to the server, usually because the server does not perform access checking.

rpc_c_authz_name
Only the client principal name is provided to the server. The server can then perform authorization based on the provided name. This form of authorization is sometimes referred to as name-based authorization.

rpc_c_authz_dce
The client's credentials (DCE Privilege Attribute Certificate or PAC) is provided to the server with each remote procedure call that is made using the binding parameter. The server performs authorization by using the client credentials. Generally, access is checked against DCE ACLs.

When a client establishes authenticated RPC, it must indicate which authorization option it wants to use.

It is the server's responsibility to implement the type of authorization appropriate for the objects that it controls. When the server calls **rpc_binding_inq_auth_caller()** to return information about an authenticated client, it gets back either the client's principal name or a pointer to the data structure that contains the client's credentials. The value that is returned depends on which type of authorization the client specified on its call to establish authenticated RPC with that server.

Each server is responsible for implementing its own access checking by means of ACL managers. When a server receives a client request for an object, the server invokes the ACL manager appropriate for that type of object and passes the manager the client's authorization data. The manager compares the client authorization data to the permissions associated with the object and either refuses or permits the requested operation. In the case of certified (PAC-based) authorization, servers must implement access checking by using the ACL facility provided by the DCE Security Service.

An ACL management API (**dce_acl***) is also available.

Name-Based Authorization

Name-based authorization (**rpc_c_authz_name**) provides a server with the client's principal name. The server call to **rpc_binding_inq_auth_caller()** retrieves the name from the binding handle associated with the client and returns it as a character string.

It is not recommended that names be used for authorization. To perform access checking using client principal names, the names must be stored in the access lists associated with the protected objects. Each time a name is changed, the change must be propagated through all the access lists in which the name is defined.

DCE Authorization

DCE authorization (**rpc_c_authz_dce**) provides a server with the client's credentials.

Credentials offer a trusted mechanism for conveying client authorization data to authenticated servers. The security service generates a client's credentials in a tamper-proof manner. When a server receives a client credentials, it knows that the credentials has been certified by DCE security.

Credentials are designed to be used with the DCE ACL facility. The ACL facility provides an editor and a set of API routines that support the implementation of access control lists and the managers to control them.

Authenticated RPC Routines

Authenticated RPC is implemented as a set of related RPC routines. Some of the routines are for use by clients, some are for use by servers and their managers, and some are for use by both clients and servers. The authenticated RPC routines are as follows:

rpc_binding_set_auth_info()
A client calls this routine to establish an authentication service, protection level, and authorization service for a server binding handle. The client identifies the server by supplying the server's principal name. The RPC runtime, in conjunction with the security service, applies the authentication service and protection level to all subsequent remote procedure calls made using the binding handle.

rpc_binding_inq_auth_info()
A client calls this routine to return the authentication service, protection level, and authorization service that are in effect for a specified server binding handle. This routine also returns the principal name of the server associated with the binding handle.

rpc_mgmt_inq_dflt_protect_level()
A client or a server calls this routine to learn the default protection level that is in force for a given authentication service.

rpc_mgmt_inq_server_princ_name()
A client, a server, or a server manager can call this routine to return the principal name that a server has registered with the RPC runtime via the **rpc_server_register_auth_info()** routine. A client can identify the desired server by supplying a server binding handle and the authentication service associated with the registered principal name.

rpc_server_register_auth_info()
A server calls this routine to register an authentication service that it wants to support and the server principal name to be associated with the registered service. The server can also optionally supply the address of a key retrieval routine to be called by the security service as part of the client authentication process. The routine is a user-supplied function whose purpose is to provide the server's key to the DCE security runtime.

Note that the server registers only an authentication service. It does not establish a protection level or an authorization service. These are the responsibilities of the client.

rpc_binding_inq_auth_caller()
A server calls this routine to return the authentication service, protection level, and authorization service that is associated with the binding handle of an authenticated client. This call also returns the server principal name specified by the client on its call to **rpc_binding_set_auth_info()**.

rpc_mgmt_set_authorization_fn()
A server calls this routine to establish a user-supplied authorization function to validate remote client calls to the server's management routines. For example, the user function can call **rpc_binding_inq_auth_caller()** to return authentication and authorization information about the calling client. The RPC runtime calls the user-supplied function whenever it receives a client request to execute one of the following server management routines:

- **rpc_mgmt_inq_if_ids()**
- **rpc_mgmt_inq_server_princ_name()**
- **rpc_mgmt_inq_stats()**
- **rpc_mgmt_is_server_listening()**
- **rpc_mgmt_stop_server_listening()**

When an unauthenticated client calls a server that has specified authentication, the RPC runtime will not perform any authentication, and *the call will reach the application manager code*. It is up to the manager to decide how to deal with the unauthenticated call.

Typically, servers and clients establish authentication as follows:

- The server specifies an authentication service for a principal identity under which it runs by calling **rpc_server_register_auth_info()**. The authentication service is specified by the *authn_svc* parameter of this call. Currently, servers may specify either DCE secret key authentication (by supplying either **rpc_c_authn_dce_secret** or **rpc_c_authn_default**) or no authentication (by supplying **rpc_c_authn_none**). The specified authentication service will be used *if it is also requested by the client*.
- The client sets authentication for a binding handle by calling **rpc_binding_set_auth_info()**. The choices are also currently either DCE secret key or no authentication. Client calls made on the binding handle attempt to use the specified authentication service.

- The server manager code calls **rpc_binding_inq_auth_caller()** to extract any authorization information from the client binding for the call.

Using RPC Within a Single Thread

The default behavior for an application client is to be single-threaded. This means that only one thread, the main thread, exists in the client process. All application and RPC runtime execution takes place within this single thread. This behavior applies only to clients that use the User Datagram Protocol (UDP). If another protocol sequence is used for RPC transport, the RPC runtime will spawn several threads and revert to multithreaded behavior.

Single-threaded behavior, compared to multithreaded client behavior, provides several benefits to application developers:

- Debugging is easier. Using advanced thread-aware debuggers and following code execution through multiple thread context switches are unnecessary. The same debugging techniques and tools used to debug standard applications can be used to debug an RPC client.
- Usage of system resources is lower. The DCE Threads runtime is not initialized in single-threaded mode. This means startup time will be faster, less memory will be used, and performance will improve because context switching does not take place.
- Linking libraries that are not thread-safe into DCE applications is less dangerous. Some third-party libraries depend on default behavior from certain operating system functions. However, in a multithreaded process this behavior is defined differently. Examples of this include signal handling, I/O, and **fork**, and **exec** functions. When an application client is single-threaded, the default behavior for these functions is guaranteed, and without risk when using libraries that are not thread-safe.

If any application-level threads are created in the RPC client, the single-threaded process immediately reverts to multithreaded behavior. This means that both the RPC runtime and DCE Threads runtime will be initialized and create several runtime-level threads, and the benefits described for a single-threaded client will no longer apply. Once the client becomes multithreaded, it remains so even if all of the user-level threads have terminated.

Existing applications can take advantage of single-threaded mode without requiring changes to the code. As long as the client is using the UDP protocol sequence and has not performed a **pthread_create** call, single-threaded behavior automatically remains; applications can continue to make pthread API calls and remain in single-threaded mode. If, for example, the application creates mutex variables, and even locks or unlocks these variables, these calls will behave correctly and not cause the process to become multithreaded. However, after the first **pthread_create** call takes place in the client application, it becomes multithreaded and all previously initialized pthreads primitives will function as expected in a multithreaded environment.

To implement single-threaded behavior, the DCE Threads library performs its initialization in two phases:

- Phase 1 occurs when the first pthread API call is made. This initializes mutexes, condition variables, and attributes.
- Phase 2 occurs when the first **pthread_create** call is made. This initializes the remaining DCE Threads functionality, including thread management, context

switching, the creation of a null background thread, and all of the multithreaded operating system behavior as described in the chapter on multithreaded programming.

Directory Services and RPC: Using the Namespace

This section discusses how the DCE RPC NSI configures directory service entries and how RPC applications can use those entries. The following topics are included:

- Directory service entries defined by NSI

  Describes the kinds of directory service entries NSI defines.
- Searching the namespace

  Describes how the namespace is searched when a client requests binding information.
- Strategies for using directory service entries

  Outlines strategies for using each kind of entry.
- The service model

  Describes the service model for defining RPC servers and introduces NSI usage models intended to guide application developers in assessing how to best use NSI for a given application.
- The resource model

  Describes the resource model for defining RPC servers.

NSI Directory Service Entries

To store information about RPC servers, interfaces, and objects, NSI defines the following directory service entries in the namespace: server entries, groups, and profiles. These directory service entries are CDS objects.

- A server entry is a directory service entry that stores binding information and object UUIDs for an RPC server.
- A group is a directory service entry that corresponds to one or more RPC servers that offer one or more RPC interfaces, type of RPC object, or both in common.
- A profile is a directory service entry that defines search paths in a namespace for a server that offers a particular RPC interface and object.

The use of server entries, groups, and profiles determines how clients view servers. A server describes itself to its clients by exporting binding information associated with interfaces and objects to one or more server entries. A group corresponds to servers that offer a given interface, service, or object. Profiles enable clients to access alternative directory service entries when searching for an interface or object. Used together, groups and profiles offer sophisticated ways for RPC applications to maintain and use directory service data.

NSI Attributes

Usually, the distinct server entries, groups, and profiles concepts are adequate for using NSI. However, the way NSI stores RPC information allows you to combine server entries, groups, and profiles into a single directory service entry. To store information about RPC applications in a directory service entry, the RPC directory service interface defines several RPC-specific directory service attributes, or NSI attributes. NSI attributes contain information about RPC applications in a directory service entry. The NSI attributes are as follows:

- NSI binding attribute

The binding attribute stores binding information and interface identifiers (interface UUID and version numbers) exported to the server entry. This attribute identifies a directory service entry as a server entry.

- NSI object attribute

  The object attribute stores a list of one or more object UUIDs. Whenever a server exports any object UUIDs to a server entry, the server entry contains an object attribute as well as a binding attribute. When a client imports from that entry, the import operation returns an object UUID from the list stored in the object attribute.

- NSI group attribute

  The group attribute stores the entry names of the members of a single group. This attribute identifies a directory service entry as an RPC group.

- NSI profile attribute

  The profile attribute stores a set of profile elements. This attribute identifies a directory service entry as an RPC profile.

Figure 34 represents the correspondence between NSI attributes and the different directory service entries: server entries, groups, and profiles.

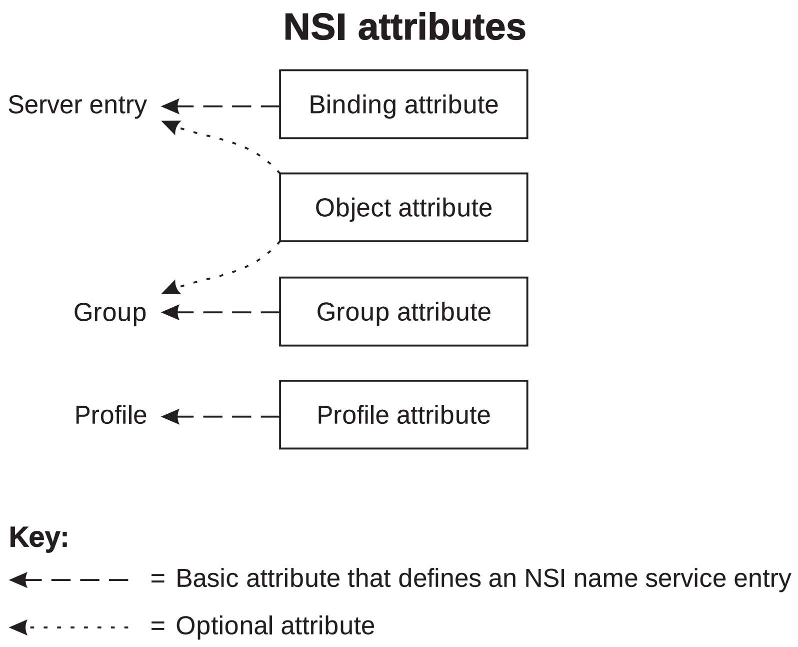


Figure 34. NSI Attributes

Any directory service entry can contain any combination of the four NSI attributes. However, to facilitate administrating directory service entries, avoid creating binding, group, and profile attributes in the same entry. Instead, use distinct directory service entries for server entries, groups, and profiles. The object attribute, in contrast, is designed as an adjunct to another NSI attribute, especially the binding attribute.

When implementing the resource model or when used to distinguish server instances, a server entry contains an object attribute as well as a binding attribute. On finding a server entry whose binding attribute contains compatible binding information, an NSI search operation also looks in the entry for an object attribute. For groups whose membership is selected according to a shared object or set of objects, it may be useful to export those objects to the group. In this case, the directory service entry of the group contains both group and object attributes. For reading the object UUIDs in the NSI object attribute in any directory service entry, NSI provides a set of object inquiry operations, called using the **rpc_ns_entry_object_inq_**{**begin**,**next**,**done**}**()** routines.

Using separate entries facilitates administration of the namespace; for example, by enabling entry names to specifically describe their contents. Keeping server entries, profiles, and groups separate allows clear references to each of them.

Note: In addition to any NSI attributes, a directory service entry contains other kinds of directory service attributes. Every entry in a namespace contains standard attributes created by the directory service. NSI operations rely on some standard attributes to identify and use an entry.

Structure of Entry Names

Each entry in a namespace is identified by a unique global name comprising a cell name and a cell-relative name.

A cell is a group of users, systems, and resources that share common DCE services. A cell configuration includes at least one cell directory server and one security server. A cell's size can range from one system to thousands of systems. A host is assigned to its cell by a DCE configuration file. For information on cells, see the *OSF DCE Administration Guide*.

The following is an example of a global name:

```
/.../C=US/O=uw/OU=MadCity/LandS/anthro/Stats_host_2
```

The parts of a global name are as follows:

- Cell name (using X.500 name syntax):

  ```
  /.../C=US/O=uw/OU=MadCity
  ```

  The symbol **/...** begins a cell name. The letters before the **=** (equal signs) are abbreviations for Country (**C**), Organization (**O**), and Organization Unit (**OU**). For entries in the local cell, the cell name can be represented by a **/.:** prefix, in place of the actual cell name; for example:

  ```
  /.:/LandS/anthro/Stats_host_2
  ```

  The **/** (slash) to the right of the cell name represents the root of the cell directory (the cell root).

  For NSI operations on entries in the local cell, you can omit the cell name.
- Cell-relative name (using DCE name syntax):

  Each directory service entry requires a cell-relative name, which contains a directory pathname and a leaf name.
  - A directory pathname follows the cell name and indicates the hierarchical relationship of the entry to the cell root.

    The directory pathname contains the names of any subdirectories in the path; each subdirectory name begins with a **/** (slash), as follows: / *sub-dir-a-name*/ *sub-dir-b-name*/ *sub-dir-c-name*

    Directory pathnames are created by directory service administrators. If an appropriate directory pathname does not exist, ask your directory service administrator to extend an existing pathname or create a new pathname. In a directory pathname, the name of a subdirectory should reflect its relationship to its parent directory (the directory that contains the subdirectory).

– A leaf name identifies the specific entry.

  The leaf name constitutes the right-hand part of a global name, beginning with the rightmost **/** (slash).

For example, **/.:/LandS/anthro/Cal_host_4**, where **/.:/** represents the cell name, **/LandS/anthro** is the directory pathname, and **/Cal_host_4** is the leaf name. If the directory service entry is located at the cell root, the leaf name directly follows the cell name; for example, **/.:/cell-profile**.

Note: When NSI is used with CDS, the cell-relative name is a CDS name.

Figure 35 shows the parts of a global name.

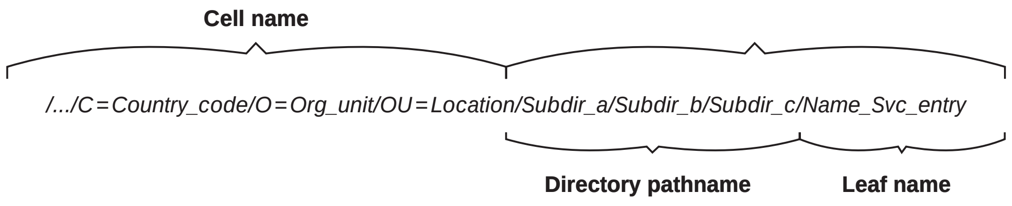

Cell name
/.../C=Country_code/O=Org_unit/OU=Location/Subdir_a/Subdir_b/Subdir_c/Name_Svc_entry
Directory pathname
Leaf name

Figure 35. Parts of a Global Name

Server Entries

NSI enables any RPC server with the necessary directory service permissions to create and maintain its own server entries in the namespace. A server can use as many server entries as it needs to advertise combinations of its RPC interfaces and objects.

Each server entry must correspond to a single server (or a group of interchangeable server instances) on a given system. Interchangeable server instances are instances of the same server running on the same system that offer the same RPC objects (if any). Only interchangeable server instances can share a server entry.

Each server entry must contain binding information. Every combination of protocol sequence and network addressing information represents a potential binding. The network addressing information can contain a network address, but lacks an endpoint, making the address partially bound.

A server entry can also contain a list of object UUIDs exported by the server. Each of the object UUIDs corresponds to an object offered by the server. In a given server entry, each interface identifier is associated with every object UUID, but with only the binding information exported with the interface.

Figure 36 on page 212 represents a server entry. This server entry was created by two calls to the **rpc_ns_binding_export()** routine. The first call created the first column of the top half of the figure. The routine's *binding_vec* parameter had three elements, each of which is paired with the routine's *if_handle* parameter. The vertical ellipsis points under the last box indicate that more elements in the routine's *binding_vec* parameter would have resulted in more interface UUID/binding information pairs in the first column.

Similarly, the second call to the **rpc_ns_binding_export()** routine created the second column of the top half of the figure. The routine's *binding_vec* parameter had two elements, each of which is paired with the routine's *if_handle* parameter. For example, the first element could have contained binding information with the *ncacn_ip_tcp* protocol sequence, and the second element could have contained binding information with the **ncadg_ip_udp** protocol sequence. As in the first column, more elements in the routine's *binding_vec* parameter would have resulted in more interface UUID/binding information pairs.

Third and subsequent calls to the **rpc_ns_binding_export()** routine would create more columns; the two pairs of horizontal ellipsis points indicate this expansion.

Finally, note that the **rpc_ns_binding_export()** routine optionally takes a vector of object UUIDs. The four object UUIDs in the bottom half of the figure came from the two calls to the routine, or from another call to the routine with no interface UUID/version and with no binding information, but with object UUIDs. The object UUIDs are associated with no particular binding. Instead, they are associated with all the bindings. Third and subsequent calls to the routine could create more object UUIDs; the vertical ellipsis points indicate this expansion.

Note: To distinguish among RPC objects when using the CDS ACL editor, export the RPC objects to separate directory service entries.

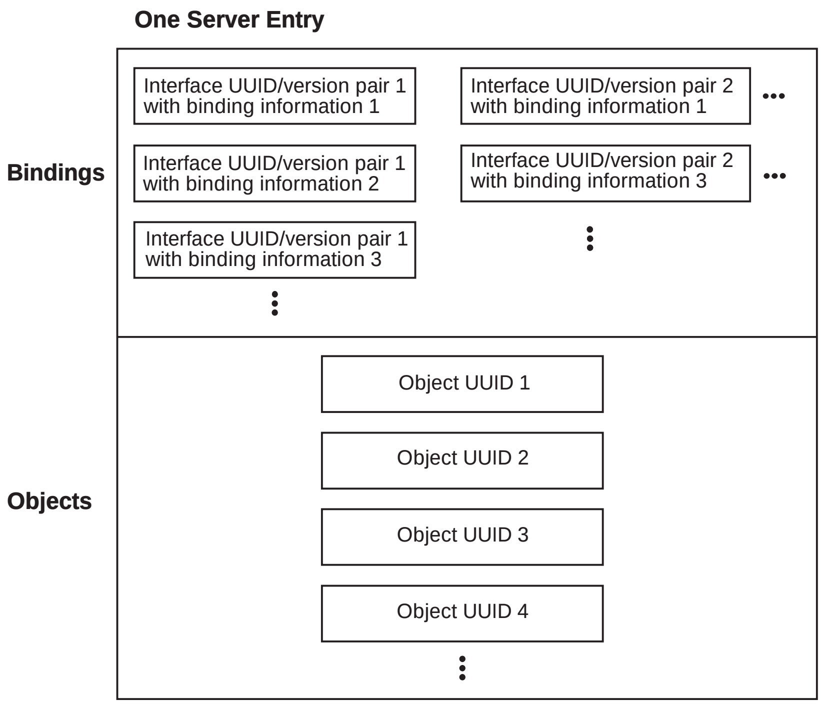

One Server Entry
Bindings
Interface UUID/version pair 1 with binding information 1
Interface UUID/version pair 2 with binding information 1
Interface UUID/version pair 1 with binding information 2
Interface UUID/version pair 2 with binding information 3
Interface UUID/version pair 1 with binding information 3
Objects
Object UUID 1
Object UUID 2
Object UUID 3
Object UUID 4

Figure 36. Possible Information in a Server Entry

Groups

Administrators or users of RPC applications can organize searches of a namespace for binding information by having clients use an RPC group as the starting point for NSI search operations. A group provides NSI search operations (**import_next** or **lookup_next** operations) with access to the server entries of different servers that offer a common RPC interface or object. A group contains names of one or more server entries, other groups, or both. Since a group can contain group names, groups can be nested. Each server entry or group named in a group is a member of the group. A group's members must offer one or more RPC interfaces, the type of RPC object, or both in common.

Figure 37 shows an example of the kinds of members a group can contain and how those members correspond to database entries.

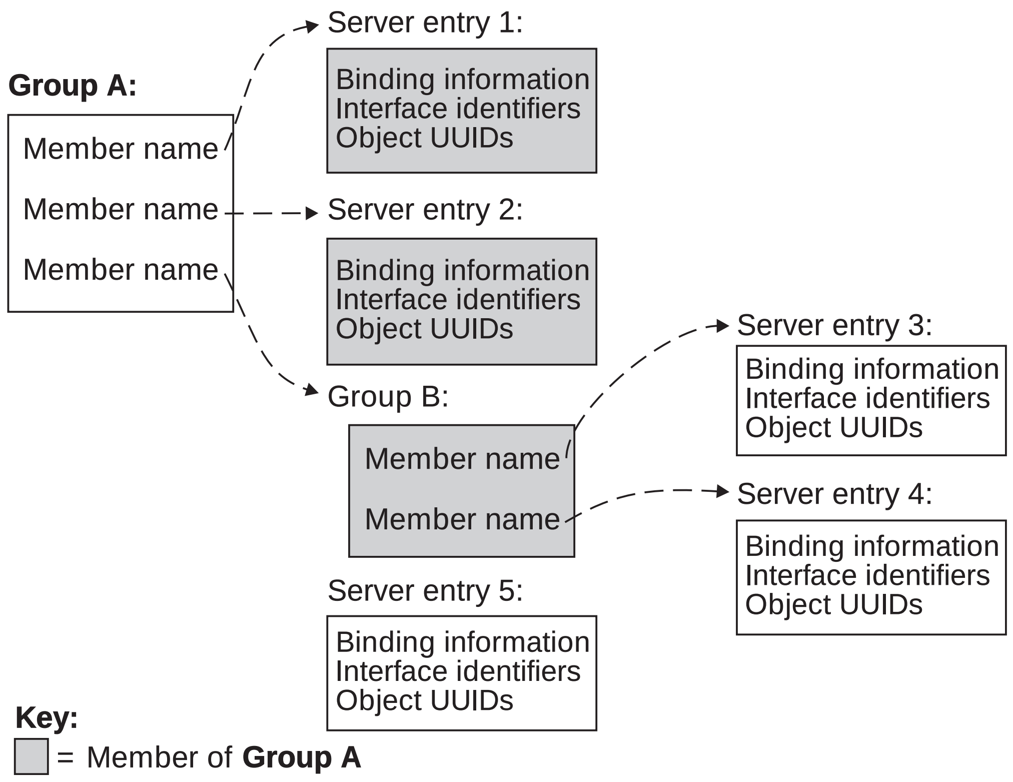


Figure 37. Possible Mappings of a Group

The members of Group A are Server Entry 1, Server Entry 2, and Group B. The members of the nested group, Group B, are Server Entry 3 and Server Entry 4. An additional server entry that advertises the common interface or object, Server Entry 5, is omitted from either group.

Profiles

Administrators or users of RPC applications can organize searches of a namespace for binding information by having clients use an RPC profile as the starting point for NSI search operations. A profile is an entry in a namespace that contains a collection of profile elements. A profile element is a database record that corresponds to a single RPC interface and that refers to a server entry, group, or profile. Each profile element contains the following information:

- Interface identifier

  This field is the key to the profile. The interface identifier consists of the interface UUID and the interface version numbers.
- Member name

  The entry name of one of the following kinds of directory service entries:
  - A server entry for a server offering the requested RPC interface
  - A group corresponding to the requested RPC interface

  - A profile
- Priority value

  The priority value (0 is the highest priority; 7 is the lowest priority) is designated by the creator of a profile element to help determine the order for using the element NSI search operations to select among like-priority elements at random.
- Annotation string

  The annotation string enables you to identify the purpose of the profile element. The annotation can be any textual information; for example, an interface name associated with the interface identifier or a description of a service or resource associated with a group.

  Unlike the interface identifier field, the annotation string is not a search key.

Optionally, a profile can contain one default profile element. A default profile element is the element that an NSI search operation uses when a search using the other elements of a profile finds no compatible binding information; for example, when the current profile lacks any element corresponding to the requested interface. A default profile element contains the nil interface identifier, a priority of 0, the entry name of a default profile, and an optional annotation.

A default profile is a backup profile, referred to by a default profile element in another profile. A profile designated as a default profile should be a comprehensive profile maintained by an administrator for a major set of users, such as the members of an organization or the owners of computer accounts on a local area network (LAN).

A default profile must not create circular dependencies between profiles; for example, when a public profile refers to an application's profile, the application's profile must not specify that public profile as a default profile.

Figure 38 on page 215 shows an example of the kinds of elements a profile can contain and how those elements correspond to database entries.

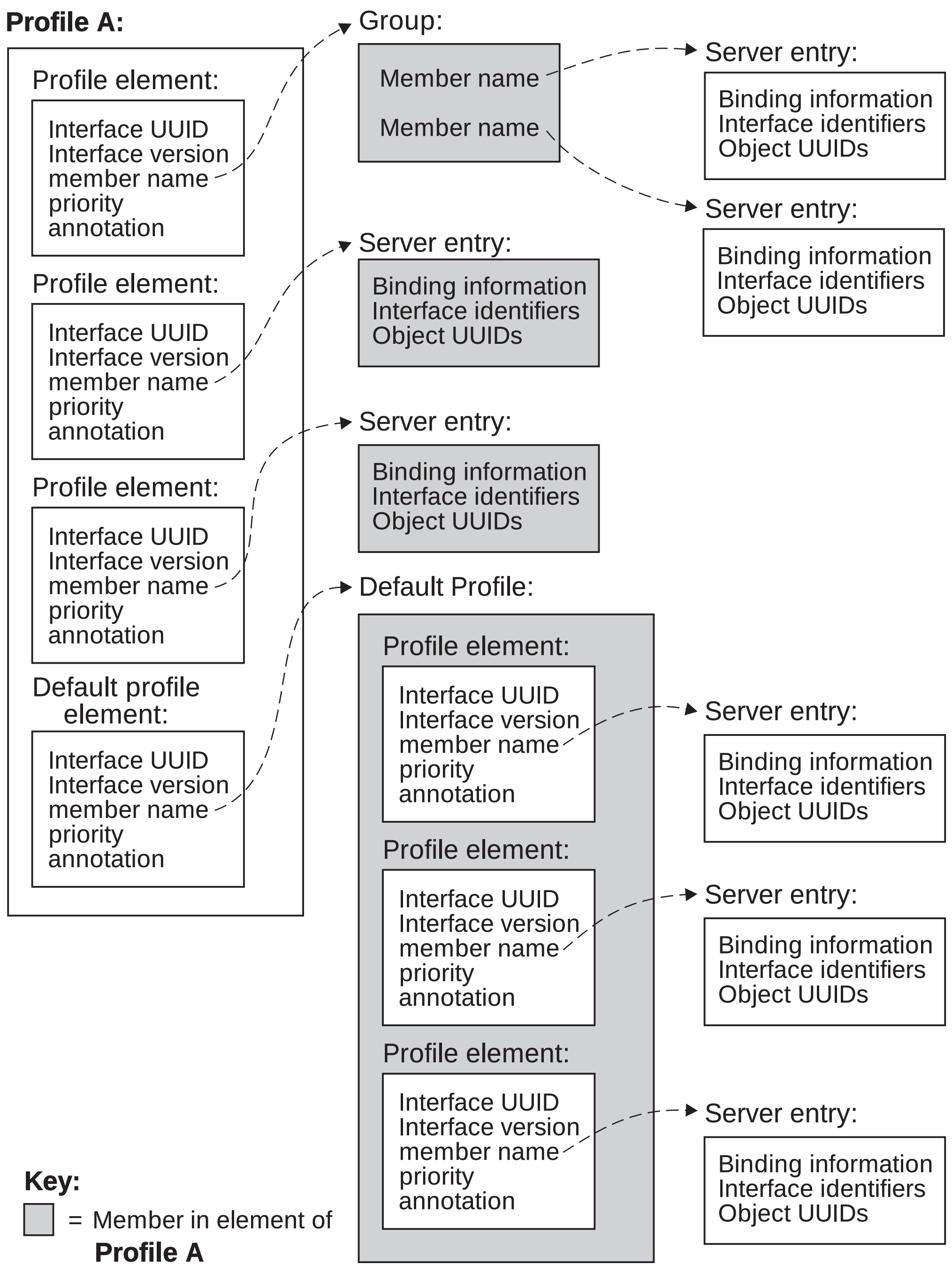


Figure 38. Possible Mappings of a Profile

NSI search operations use a profile to construct an NSI search path. When an NSI search operation reads a profile, the operation dynamically constructs its NSI search path from the set of elements that correspond to a common RPC interface.

A profile element is used only once per NSI search path. The construction of NSI search paths depends partly on the priority rankings of the elements. A search operation uses higher-priority elements before lower-priority elements. Elements of equal priority are used in random order, permitting some variation in the NSI search paths between searches for a given interface. If nondefault profile elements do not satisfy a search, the search path extends to the default profile element, if any.

Profiles meet the needs of particular individuals, systems, LANs, sites, organizations, and so forth, with minimal configuration management. The administrator of a profile can set up NSI search paths that reflect the preferences of the profile's user or users. The profile administrator can set up profile elements that refer (directly or indirectly) to only a subset of the server entries that offer a given RPC interface. Also, the administrator can assign different search priorities to the elements for an interface.

Guidelines for Constructing Names of Directory Service Entries

A global name includes both a cell name and a cell-relative name composed of a directory pathname and a leaf name. The cell name is assigned to a cell root at its creation. When you specify only a cell-relative name to an NSI operation, NSI automatically expands the name into a global name by inserting the local cell name. When returning the name of a directory service entry, a group member, or a member in a profile element, NSI operations return global names.

The directory pathname and leaf name uniquely identify a directory service entry. The leaf name should somehow describe the entry; for example, by identifying its owner or its contents. The remainder of this section contains guidelines for choosing leaf names.

Note: Directory pathnames and leaf names are case sensitive.

Use the following guidelines for constructing names:

- Naming a server entry

  For a server entry that advertises an RPC interface or service offered by a server, the leaf name must distinguish the entry from the equivalent entries of other servers. When a single server instance runs on a host, you can ensure a unique name by combining the name of the service, interface (from the interface definition), or the system name for the server's host system.

  For example, consider two servers, one offering a calendar service on host **JULES**, and one on host **VERNE**.

  The server on **JULES** uses the following leaf name:

  ```
  calendar_JULES
  ```

  The server on **VERNE** uses the following leaf name:

  ```
  calendar_VERNE
  ```

  For servers that perform tasks on or for a specific system, an alternative approach is to create server entries in a system-specific host directory within the namespace. Each host directory takes the name of the host to which it corresponds. Because the directory name identifies the system, the leaf name of the server entry name does not need to include the host name, for example:

  ```
  /.:/LandS/host_1/Process_control
  ```

  To construct names for the server entries used by distinctive server instances on a single host, you can construct unique server entry names by combining the following information: the name of the server's service, interface, or object; the system name of the server's host system; and a reusable instance identifier such as an integer.

For example, the following leaf names distinguish two instances of a calendar service on the **JULES** system:

```
calendar_JULES_01
```

```
calendar_JULES_02
```

Avoid automatically generating entry names for the server entries of server instances; for example, by using unique data such as a timestamp (**calendar_verne_15OCT91_21:25:32**) or a process identifier (**calendar_jules_208004D6**). When a server incorporates such unique data into its server entry names, each server instance creates a separate server entry, causing many server entries. When a server instance stops running, it leaves an obsolete server entry that is not reused. The creation of a new entry whenever a server instance starts may impair performance.

A server can use multiple server entries to advertise different combinations of interfaces and objects. For example, a server can create a separate server entry for a specific object, and the associated interfaces. The name of such a server entry should correspond to a well-known name for the object. For example, consider a server that offers a horticulture bulletin board known to users as **horticulture_bb**. The server exports the **horticulture_bb** object, binding information, and the associated bulletin-board interface to a server entry whose leaf name identifies the object, as follows:

```
horticulture_bb
```

Note: An RPC server that uses RPC authentication can choose identical names for its principal name and its server entry. Use of identical names permits a client that calls the **rpc_binding_set_auth_info()** routine to automatically determine a server's principal name. (The client will assume the principal name to be the same as the server's entry name.) If a server uses different principal and server entry names, users must explicitly supply the principal name. For an explanation of principal names, see "Part 5. DCE Security Service" on page 481 of this guide.

- Naming a group

  The leaf name of a group should indicate the interface, service, or object that determines membership in the group. For example, for a group whose members are selected because they advertise an interface named **Statistics**, the following is an effective leaf name:

  ```
  Statistics
  ```

  For a group whose members advertise laser printer print queues as objects, the following is an effective leaf name:

  ```
  laser-printer
  ```

- Naming a profile

  The leaf name of a profile should indicate the profile users; for example, for a profile that serves the members of an accounting department, the following is an effective leaf name:

  ```
  accounting_profile
  ```

The following text describes the NSI **begin**, **next**, and **done** operations. NSI accesses a variety of search and inquire operations that read NSI attributes in directory service entries. An NSI attribute is an RPC-defined attribute of a directory service entry used by the DCE RPC directory service interface. An NSI attribute stores one of the following: binding information, object UUIDs, a group, or a profile. Reading information from any attribute involves an equivalent set of search or inquire operations; that is, an integral set of **begin**, **next**, and **done** operations. An RPC application uses these operations as follows:

1. The application creates a directory service handle (a reference to the context of the ensuing series of **next** operations) by calling an NSI **begin** operation.
2. The application calls the NSI **next** operation corresponding to the **begin** operation one or more times. Each **next** operation returns another value or list of values from the target RPC directory service attribute. For example, an **import_next** operation returns binding information from a binding attribute and an object from an object attribute.
3. The application deletes the directory service handle by calling the corresponding NSI **done** operation.

Note: Search and inquire operations are also accessible interactively from within the RPC control program.

Table 11 lists the NSI **next** operations used by RPC applications.

Table 11. NSI next Operations

Search Operation	**Attributes Traversed**
rpc_ns_binding_import_next()	Searches for binding and object attributes of a compatible server; reads any NSI attribute in a search path. Returns a binding handle that refers to a potential binding for a compatible server, and also to a single object UUID.
rpc_ns_binding_lookup_next()	Searches for binding and object attributes of a compatible server; reads any NSI attribute in a search path. Returns a list of binding handles, each of which refers to a potential binding for a compatible server, and also to a single object UUID. The same object UUID is associated with each potential binding. Note that, after calling the **lookup_next** operation, the client must select one binding handle from the list. To select a binding handle at random, the client can call the NSI binding select routine **rpc_ns_binding_select()**. For an alternative selection algorithm, the client can define and call its own application-specific select algorithm.
Inquire Operation	**Attributes Traversed**
rpc_ns_group_mbr_inq_next()	Reads a group attribute and returns a member name.
rpc_ns_profile_elt_inq_next()	Reads a profile attribute and returns the fields of a profile element.

Selecting the Starting Entry

When searching a namespace for an RPC interface and object, a client supplies the name of the directory service entry where the search begins. The entry can be a server entry, group, or profile. Generally, an NSI search starts with a group or profile. The group or profile defines a search path that ends at a server entry containing the requested interface identifier, object UUID, and compatible binding information.

A user may know in advance what server instance to use. In this case, starting with a server entry for the server instance is appropriate.

Environment Variables

DCE RPC provides predefined environment variables that a client can use for NSI operations. An environment variable is a variable that stores information, such as a name, about a particular environment. The NSI interface provides two environment variables, **RPC_DEFAULT_ENTRY** and **RPC_DEFAULT_ENTRY_SYNTAX**, which are described in the *OSF DCE Application Development Reference*. Used together, these environment variables identify an entry name and indicate its syntax.

When a client searches for binding information, the search starts with a specific entry name. Optionally, a client can specify this entry name as the value of the **RPC_DEFAULT_ENTRY** variable. A client can also specify the name syntax of the starting entry as the value of the **RPC_DEFAULT_ENTRY_SYNTAX** variable; the default name syntax is **dce**.

Note: The **dce** name syntax is the only syntax currently supported by CDS. However, NSI is independent of any specific directory service and depending on your vendor, may support one or more alternative directory services that use different name syntaxes.

Searching the Namespace for Binding Information

Searching the namespace for binding information requires that a client specify a starting point for the search. A client can start with a specific server entry. However, this is a limiting approach because the client is restricted to using one server. To avoid this, a client can start searching with a group or a profile instead of with a server entry. Searches that start with a profile or a group should encounter the server entry of a compatible server. If such an entry is not encountered, a search operation returns the **rpc_s_no_more_bindings** status code to the client. When calling the routines **rpc_ns_binding_import_next()** or **rpc_ns_binding_lookup_next()**, a client must track whether the routine returns this status code.

The import_next and lookup_next Search Algorithm

The NSI search operations (**import_next** and **lookup_next**) traverse one or more entries in the namespace when searching for compatible binding information. In each directory service entry, these operations ignore non-RPC attributes and process the NSI attributes in the following order:

1. Binding attribute (and object attribute, if present)
2. Group attribute
3. Profile attribute

If an NSI search path includes a group attribute, the search path can encompass every entry named as a group member. If a search path includes a profile attribute, the search path can encompass every entry named as the member of a profile element that contains the target interface identifier. A search finishes only when it finds a server entry containing compatible binding information and the nonnil object UUID, if requested. Search operations take the following steps when traversing a directory service entry:

Step 1:
Binding attribute

In each entry, the search operation starts by searching for a compatible interface identifier in the binding attribute, if present.

The absence of a binding attribute or of any compatible interface identifier causes the search operation to go directly to step 2.

The presence of any compatible interface identifier indicates that compatible potential bindings may exist in the binding attribute. At this point, object UUIDs may impact the search, as follows:

- If the client specified the nil object UUID, object UUIDs do not affect the success or failure of the search. The search returns compatible binding information for one or more potential bindings.
- If the client specified a nonnil object UUID, the search reads the object attribute, if present, to look for the requested object UUID. This search for an object UUID has one of the following outcomes:
  - On finding the specified object UUID, the search returns the object UUID along with compatible binding information for one or more potential bindings.
  - If a requested object UUID is absent, the search continues to step 2.

Note: If a search involves a series of **import_next** or **lookup_next** operations, a subsequent next operation resumes the search at the point in the search path where the preceding operation left off.

Step 2:
Group attribute

If the binding attribute does not lead to compatible binding information or if a series of **import_next** or **lookup_next** operations exhausts the compatible binding information, the search continues by reading the group attribute, if present; if the directory service entry lacks a group attribute, the search goes directly to step 3.

The search operation selects a member of the group at random, goes to the entry of that member, and resumes the search at step 1. Unless a group member leads the search to compatible binding information, the search looks at all the members of the group, one by one in random order, until none remain.

Step 3:
Profile attribute

If the binding and group attributes do not lead to compatible binding information, the search continues by reading the profile attribute, if present; if the directory service entry lacks a profile attribute, the search fails.

The search operation identifies all the profile elements containing the requested interface identifier and searches them in the order of their priority, beginning with the 0 (zero) priority elements. Profile elements of a given priority are searched in random order. For the selected profile element, the search reads the member name and goes to the corresponding directory service entry. There, the search resumes at step 1. Unless a profile element leads the search to compatible binding information, the search eventually looks at all the profile elements with the requested interface identifier, one by one, until none remain.

If the starting entry does not contain NSI attributes, or if none of the steps satisfies the search, the search operation returns the status code **rpc_s_no_more_bindings** to the client.

Note: The inquire next (**inq_next**) operations for objects, groups, or profiles look at only the entry specified in its corresponding inquire begin (**inq_begin**) operation. The search ignores nested groups or nested profiles.

Figure 39 on page 222 illustrates the three steps of the **import_next** and **lookup_next** search operations.

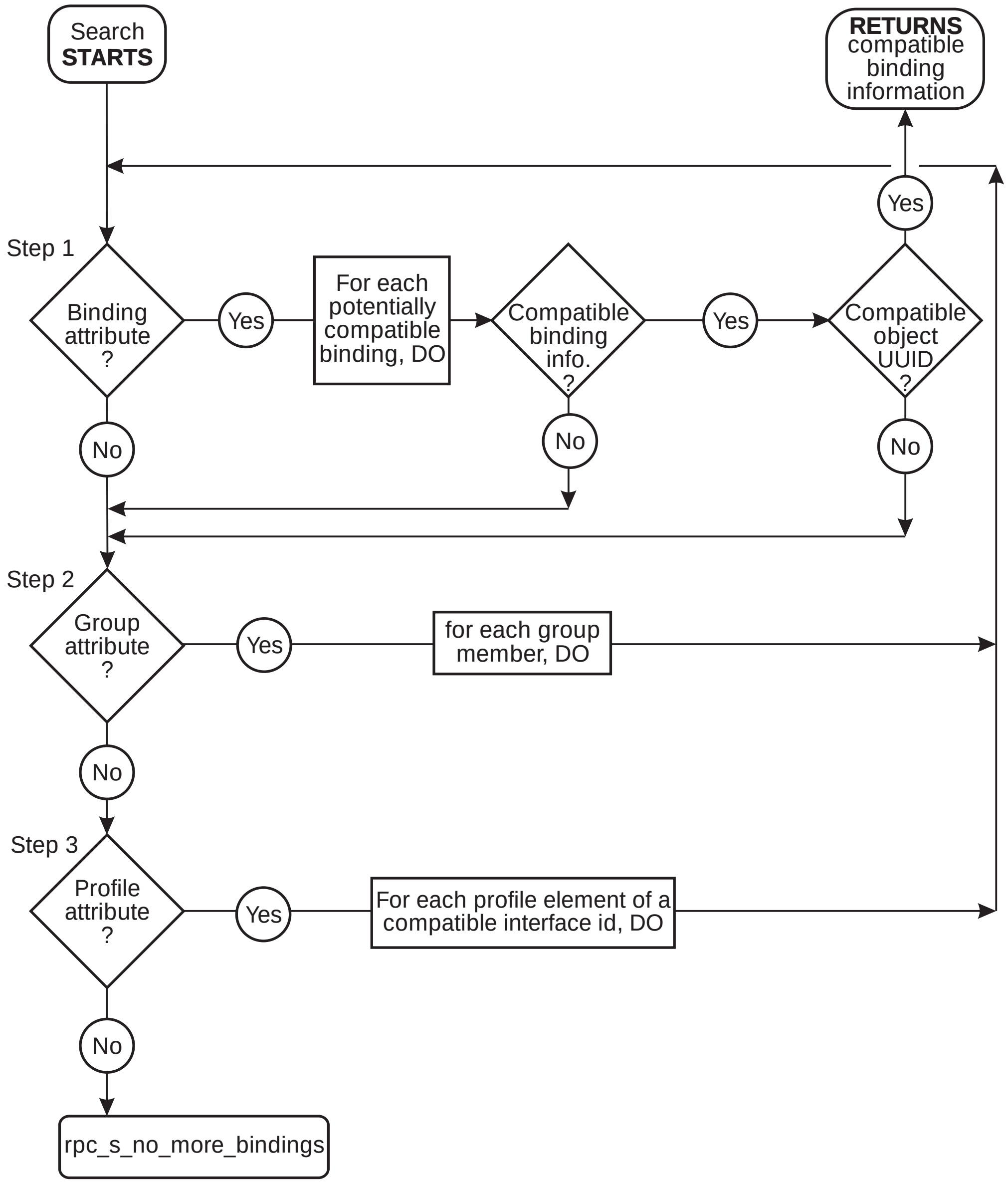

Search STARTS
RETURNS compatible binding information
Yes
Step 1
Binding attribute ?
Yes
For each potentially compatible binding, DO
Compatible binding info. ?
Yes
Compatible object UUID ?
No
No
No
Step 2
Group attribute ?
Yes
for each group member, DO
No
Step 3
Profile attribute ?
Yes
For each profile element of a compatible interface id, DO
No
rpc_s_no_more_bindings

Figure 39. The import_next, lookup_next Search Algorithm in a Single Entry

Examples of Searching for Server Entries

This subsection provides several examples of how the NSI **import_next** and **lookup_next** operations search for binding information associated with a given RPC interface and object in a namespace.

The examples in this guide use the following conventions:

- To simplify the following examples, each member name is represented by a leaf name preceded by the symbol that represents the local cell (**/.:**). For example, the full global name of the group for the **Bulletin_board_interface** is as follows:

```
/.../C=US/O=uw/OU=MadCity/LandS/bb_grp
```

The abridged name is **/.:/LandS/bb_grp**.

- Except for the nil interface UUID of the default profile, the examples avoid string representations of actual UUIDs. Instead, the examples represent a UUID as a value consisting of the name of the interface and the string *if-uuid* or of the name of the object and the string *object-uuid*; for example:

  ```
  calendar-if-uuid,1.0

  laser-printer-object-uuid
  ```
- Profile elements in the examples are organized as follows (annotations are not displayed):

  ```
  interface-identifier  member-name  priority
  ```

  For example,

  ```
  2fac8900-31f8-11ca-b331-08002b13d56d,1.0 /.:/LandS/C_host_7 0
  ```

  which, in the following examples, is represented as follows:

  ```
  calendar-if-uuid,1.0  /.:/LandS/C_host_7 0
  ```

  Note: The priority is a value of 0 to 7, with 0 having the highest search priority and 7 having the lowest priority.

The first two examples begin with the personal profile of a user, Esther Rose, whose user name is **esther_r** and whose profile has the leaf name of **esther_r_profile**. To use this profile, Esther must specify its entry name to the client. Usually, a client either uses the predefined RPC environment variable **RPC_DEFAULT_ENTRY** or prompts for an entry name. For a client to use **RPC_DEFAULT_ENTRY**, the client or user must have already set the variable to a directory service entry.

The following example illustrates six profile elements from the individual user profile used in the first two examples. The six elements include five nondefault elements for some frequently used interfaces and a default profile element. Each profile element is displayed on three lines, but in an actual profile all the fields occupy a single record. The fields are the interface identifier (interface UUID and version numbers), member name, priority, and annotation.

```
/.:/LandS/anthro/esther_r_profile contents:

ec1eeb60-5943-11c9-a309-08002b102989,1.0
  /.../C=US/O=uw/OU=MadCity/LandS/Cal_host_7
  0 Calendar_interface_V1.0

ec1eeb60-5943-11c9-a309-08002b102989,2.0
  /.../C=US/O=uw/OU=MadCity/LandS/Cal_host_4
  1 Calendar_interface_V2.0

62251ddd-51ed-11ca-852c-08002b1bb4f6,2.0
  /.../C=US/O=uw/OU=MadCity/bb_grp
  0 Bulletin_board_interface_V2.0

62251ddd-51ed-11ca-852c-08002b1bb4f6,2.1
  /.../C=US/O=uw/OU=MadCity/bb_grp
  1 Bulletin_board_interface_V2.1

9e18d295-51ec-11ca-9cc0-08002b1bb4f5,1.0
  /.../C=US/O=uw/OU=MadCity/LandS/anthro/Zork_host_2
```

```
  0 Zork_interface_V1.0

00000000-0000-0000-0000-000000000000,0.0
  /.../C=US/O=uw/OU=MadCity/cell-profile
  0 Default_profile_element
```

Example 1: Importing for an Interface with Multiple Versions: **Target Interface:** Calendar V2.0

1. The search for binding information associated with Calendar V2.0 starts with the entry **esther_r_profile**:

   ```
   /.../C=US/O=uw/OU=MadCity/LandS/anthro/esther_r_profile contents:

     calendar-if-uuid,1.0  /.:/LandS/C_host_7  0
     calendar-if-uuid,2.0  /.:/LandS/C_host_4  1

     bulletin_board-if-uuid,2.0  /.:/LandS/bb_grp  2
     bulletin_board-if-uuid,2.1  /.:/LandS/bb_grp  3

     Zork-if-uuid,1.0  /.:/Eng/Zork_host_2  0
     00000000-0000-0000-0000-000000000000,0.0  /.:/cell-profile 0
   ```

   The search operation examines only the two profile elements that refer to the Calendar interface:

   a. The operation rejects the first profile element for the interface because it refers to the wrong version numbers.
   b. In the next profile element, the operation finds the correct version numbers (**2.0**). The search proceeds to the associated server entry, **/.:/LandS/Cal_host_4**.
2. The search ends with the indicated server entry, where the binding information requested by the client resides:

   ```
   /.:/LandS/Cal_host_4 contents:
     calendar-if-uuid,2.0


    binding-information
   ```

Example 2: Using a Default Profile for Importing an Interface: **Target Interface:** Statistics V1.0

1. The search for binding information associated with Statistics V1.0 starts with the entry **esther_r_profile**. But the profile lacks any elements for the interface. Thus the search reaches the default profile element, which provides the entry name for the default profile, **/.:/cell-profile**:

   ```
   /.:/LandS/anthro/esther_r_profile contents:

     calendar-if-uuid,1.0  /.:/LandS/C_host_7  0
     calendar-if-uuid,2.0  /.:/LandS/C_host_4  1

     bulletin_board-if-uuid,2.0  /.:/LandS/bb_grp  2
     bulletin_board-if-uuid,2.1  /.:/LandS/bb_grp  3

     Zork-if-uuid,1.0  /.:/Eng/Zork_host_2  0
     00000000-0000-0000-0000-000000000000,0.0  /.:/cell-profile
   0
   ```
2. The search continues to the indicated default profile, **/.:/cell-profile**, which contains a profile element for the requested Statistics V1.0 interface:

```
/.:/LandS/cell-profile contents:
 .
 .
 .
 Statistics-if-uuid,1.0  /.:/LandS/Stats_host_6  0
 .
 .
 .
```

3. The search ends at the indicated server entry, **/.:/LandS/Stats_host_6**, where a server address for the requested interface resides:

```
/.:/LandS/Stats_host_6 contents:

  Statistics-if-uuid,1.0

  binding-information
```

Example 3: Importing an Interface and an Object: **Target Interface:** Print Server V2.1

Target Object: Laser Printer Print Queue

1. The search starts with the entry **/.:/Bldg/Print_queue_grp**, which contains the entry names of several server entries that advertise the **Print_server** interface and the object UUID of a given **Laser_printer** print queue. The search begins by randomly selecting a member name. In this instance, the search selects the name **/.:/Bldg/Print_server_host_3**:

```
/.:/Bldg/Print_queue_grp contents:

  /.:/Bldg/Print_server_host_3
  /.:/Bldg/Print_server_host_7
  /.:/Bldg/Print_server_host_9
```

2. The search continues with the **/.:/Bldg/Print_server_host_3** entry. There, it finds the requested Version 2.1 of the **Print_server** interface. However, the search continues because the entry lacks the object UUID of the requested **Laser_printer** queue:

```
/.:/Bldg/Print_server_host_3 contents:

 print_server-if-uuid,2.1

 binding-information

 line_printer_queue-object-uuid
```

3. The search goes back to the previous entry, which was **/.:/Bldg/Print_queue_grp**, to select another entry name; in this instance **/.:/Bldg/Print_server_host_9**:

```
/.:/Bldg/Print_queue_grp contents:

  /.:/Bldg/Print_server_host_3
  /.:/Bldg/Print_server_host_7
  /.:/Bldg/Print_server_host_9
```

4. The search selects the **/.:/Bldg/Print_server_host_9** entry. This entry contains both a server address for the requested Version 2.1 of the interface and the requested object UUID of the **Laser_printer** queue:

```
/.:/Bldg/Print_server_host_9 contents:

 print_server-if-uuid, 2.1

 binding-information

 laser_printer_queue-object-uuid
```

The search returns binding information from this entry to the client.

Expiration Age of a Local Copy of Directory Service Data

To prevent accessing a namespace unnecessarily, previously requested directory service data is sometimes stored on the system where the request originated. A local copy of directory service data is not automatically updated at each request. Automatic updating of the local copy occurs only when it exceeds its expiration age. The expiration age is the amount of time that a local copy of directory service data from an NSI attribute can remain unchanged before a request from an RPC application for the attribute requires updating of the local copy. When an RPC application begins running, the RPC runtime randomly specifies a value between 8 and 12 hours as the default expiration age for that instance of the application. Most applications use only this default expiration age, which is global to the application.

An expiration age is used by an NSI next operation, which reads data from directory service attributes. For a given search or inquire operation, you can override the default expiration age by calling the routine **rpc_ns_mgmt_handle_set_exp_age()** after the operation's begin routine. Note that specifying a low default age will result in increased network updates among the name servers in your cell. This will adversely affect the performance of all network traffic. Therefore, use the default whenever possible. If you must override the default age, specify a number that is high enough to avoid frequent updates of local data.

An NSI next operation usually starts by looking for a local copy of the attribute data being requested by an application. In the absence of a local copy, the NSI next operation creates one with fresh attribute data from the namespace. If a local copy already exists, the operation compares its actual age to the expiration age used by the application. If the actual age exceeds the expiration age, the operation automatically tries to update the local copy with fresh attribute data. If updating is impossible, the old local data remains in place and the NSI next operation fails, returning the **rpc_s_name_service_unavailable** status code.

Strategies for Using Directory Service Entries

When developing an RPC application, decide how an application will use the namespace and design your application accordingly. The following subsections discuss issues associated with how servers use different types of directory service entries.

Using Server Entries

An application requires separate server entries for servers on different hosts. For example, if a server offering the calendar service runs on two hosts, **JULES** and **VERNE**, one server entry is necessary for the server on **JULES** and another is necessary for the server on **VERNE**.

Each server entry requires a unique cell-relative entry name. If a server adheres to a simple and consistent arrangement of server entries, you may be able to use server initialization code to automatically generate a name for each server entry, and also to ensure that the entry exists. However, some servers will need to obtain the entry name of a server entry from an external source such as a command-line argument or a local database belonging to the application.

Note: Applications that obtain entry names and UUIDs as command-line arguments should accept user-defined values that represent them as an alternative to accepting the actual names.

Some applications, such as a process-control application, require only one server instance per system. Many applications, however, can accommodate multiple server instances on a system. When multiple instances of a server run simultaneously on a single system, all instances on a host can use a single server entry, every instance can use separate server entries, or the instances can be classified into subsets with a separate server entry. A client importing from a shared server entry cannot distinguish among the server instances that export to the entry. Therefore, the recommended strategy for a server on a given system depends on which server instances are viewed by clients as interchangeable entities and which are viewed as unique entities, as follows:

- Interchangeable server instances

  When clients consider all the server instances on a host as equivalent alternatives, all of the instances can (and should) share a server entry. For example, multiple instances of the calendar service running on host **JULES** can all export to the **calendar_JULES** entry.
- Unique server instances

  A unique server instance possesses a significant difference from other instances of the same host. Unique server instances require separate server entries. Each server instance must export unique information to its own server entry; this unique information can be either a server-specific, well-known endpoint or an object UUID belonging exclusively to the one server instance.

  Before exporting, each server instance must acquire the entry name of its server entry from an external source. When a unique server instance stops running, its server entry becomes available. An available server entry should be reused for a new instance of that server by providing the existing entry's name for a new server instance to use with the export operation. If any existing server entries are unavailable, a new server instance requires a new server entry name.

  For a discussion of when a server instance should remove the binding information from its server entry, see the **rpc_ns_binding_unexport(3rpc)** reference page.

Using Groups

When a server is first installed on a system, the server or the installer creates one or more server entries for the server. Also, when installing the first instance of the server within a cell, the installer usually creates one or more groups for the application. For any application, the local system and directory service administrators can create site-specific groups whose members are server entries, groups, or both. Typically, a server adds a server entry to at least one group.

Design decisions for defining groups may reflect a number of possible factors. Typical factors that help define effective groups include the proximity of services or resources to clients, the types of any resources offered by servers, the uses of UUIDs, and the types of users that require a specific server.

For example, for a print server, proximity to the clients and the type of supported file formats are both relevant. These factors may affect print servers as follows:

- Proximity

  If the proximity of a server is important to clients, assign servers to groups according to their locations. For example, print servers that are located together can use their own group (for example, print servers in building 1 use the group **bldg_1_print_servers**). Each server instance can add its own entry to the group, or a system administrator can add server entries by using the RPC control program.

  To select randomly among servers in a given location, a client imports using the name of a group that corresponds to those servers (or of a profile that refers to that group).

  Note: If proximity is the key factor in selecting among servers, name each server entry for the server's location; for example, **bldg_1_pole_27_print_server**.

- Object types

  When accessing specific classes of resources is important to clients, you can group server instances based on the type of object they offer.

  For servers that advertise resources in server entries, groups often use subsets for server entries according to the resources they advertise. For example, print servers can be grouped according to supported file formats. In this case, an administrator creates a group entry for each file format; for example, **post_printers**, **sixel_printers**, and **ascii_printers**. Each print server entry is a member of one or more groups.

  Users that specify a group for a file format must find the printer that processes the print command. To help the user find the printer, the client can obtain the name of the server entry that supplied the server binding information by calling **rpc_ns_binding_inq_entry_name()**, and then display the name for the user. If the server entry name indicates the location of the print server (for example, **floor_3_room_45A_print_server**), the user can then find the printer.

An application can set up groups according to different factors for different purposes. For example, the print server application can set up groups of neighboring print servers and a group of print servers for each of the file formats. The same server is a member of at least one group of each kind. Clients require users to specify the name of a directory service entry as a command-line argument of remote print commands. The user specifies the name of the appropriate group.

Note: If a user wants a specific print server and knows the name of its server entry, the user can specify that name to the client instead of a group.

Using Profiles

Profiles are tools for managing NSI searches (performed by **import_next** or **lookup_next** operations). Often profiles are set up as public profiles for the users of a particular environment, such as a directory service cell, a system, a specific application, or an organization. For example, the administrator of the local directory service cell should set up a cell profile for all RPC applications that use the cell, and the administrator of each system in the distributed computing environment should set up a system profile for local servers.

For each application, a directory service administrator or the owner of an application should add profile elements to the public profiles that serve the general user population; for example, a cell profile, a system profile, or a profile of an organization. Each profile element associates a profile member (represented in the member field of an element as the global name of a directory service entry) with an interface identifier, access priority, and optional annotation. A candidate for membership in a cell profile is a group or another profile; for example, a group that refers, directly or indirectly, to the servers of an application installed in the local cell or an application-specific profile.

An application may benefit from an application-specific profile. For example, an administrator at a specific location, such as a company's regional headquarters, can assign priorities to profile elements based on the proximity of servers to the headquarters, as illustrated by Figure 40.

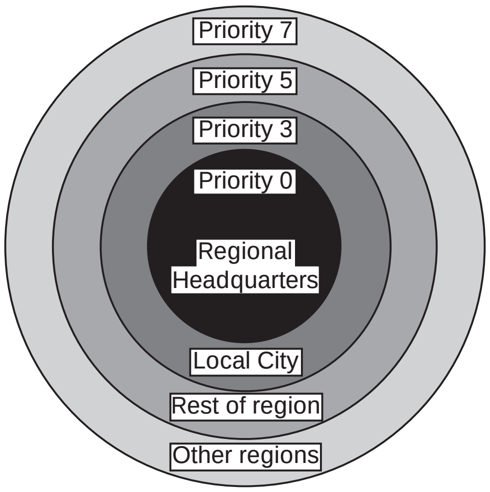


Figure 40. Priorities Assigned on Proximity of Members

An individual user can have a personalized user profile that contains elements for interfaces the user uses regularly and a default element that specifies a public profile, such as the cell profile, as the default profile. NSI searches use the default profile when a client needs an RPC interface that lacks an element in the user profile.

The Service Model for Defining Servers

The NSI operations accommodate two distinct models for defining servers: the service model and the resource model. These models express different views of how clients use servers and how servers can present themselves in the directory service database. The models are not mutually exclusive, and an application may need to implement both models to meet diverse goals. By evaluating these models before designing an RPC application, you can make informed decisions about whether and how to use object UUIDs, how many server entries to use per server, how to distinguish among instances of a server on a system, whether and how to use groups or profiles or both, and so forth. The two models are the service model and resource model.

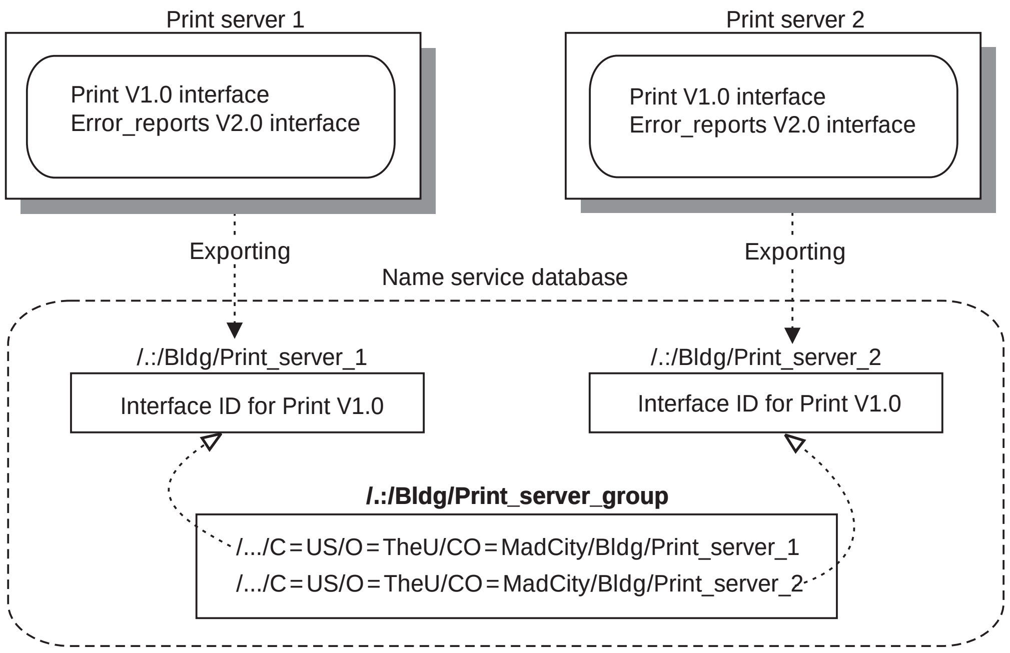
The service model views a server exclusively as a distributed service composed of one or more application-defined interfaces that meet a common goal independently of specific resources. The service model is used by applications whose servers offer an identical service and whose clients do not request an RPC resource when importing an interface. Often, with the service model, all the server instances of an application are equivalent and are viewed as interchangeable. However, the service model can accommodate applications that view each server instance as unique. The implications of whether server instances are viewed as interchangeable or unique are significant, so the following subsections address these alternatives separately.

Interchangeable Server Instances

With the service model, servers offer an identical service that operates the same way on all host systems. For example, an application that uses the service model is a collection of equivalent print servers that support an identical set of file formats, and that are installed on printers in a single location. The print servers in any location can be segregated from printer servers elsewhere by using a location-specific group.

Figure 41 shows interchangeable print servers offering an identical print service on different hosts. To access this service, clients request the Print V1.0 interface and specify the nil object UUID. In this illustration, the starting entry for the NSI search is a group corresponding to local print servers. Note that a client may be able to reach this print server group by starting from a profile or another group.

Note: To simplify the illustrations of the usage models, the contents of server entries are represented without listing any binding information.



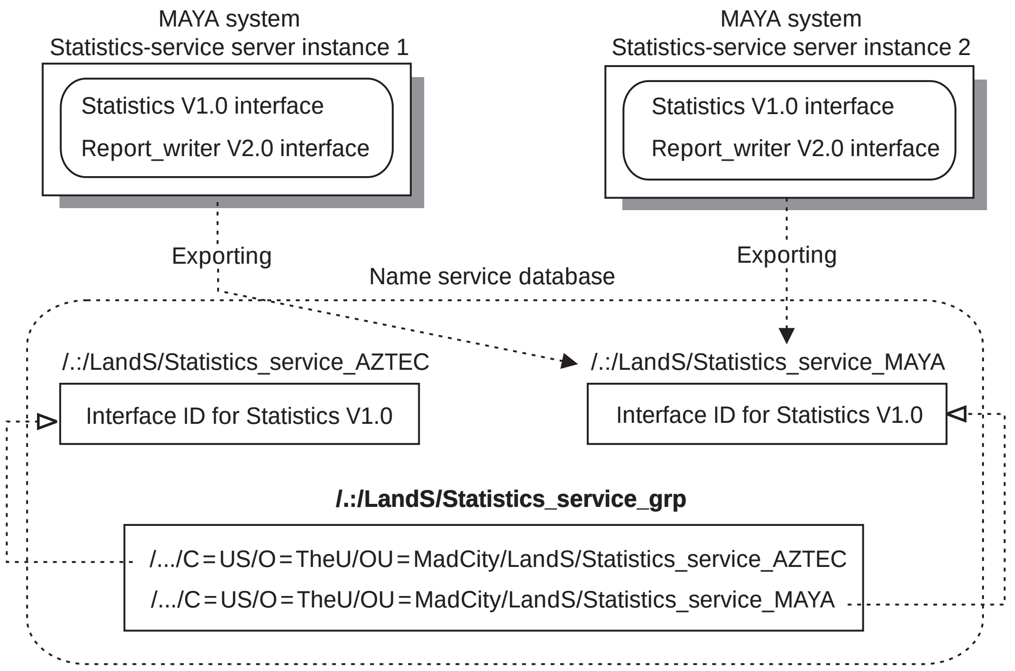
Print server 1
Print V1.0 interface
Error_reports V2.0 interface
Print server 2
Print V1.0 interface
Error_reports V2.0 interface
Exporting
Exporting
Name service database
/.:/Bldg/Print_server_1
Interface ID for Print V1.0
/.:/Bldg/Print_server_2
Interface ID for Print V1.0
/.:/Bldg/Print_server_group
/.../C=US/O=TheU/CO=MadCity/Bldg/Print_server_1
/.../C=US/O=TheU/CO=MadCity/Bldg/Print_server_2

Search Requirements
Target Interface: Printer V1.0
Target Object: None
Starting Entry: /.:/Bldg/Print_server_group
Maximum number of traversed entries: 2

Figure 41. Service Model: Interchangeable Instances on Two Hosts

Note: The number of entries traversed by a search operation is unrelated to the number of binding handles it returns.

Figure 42 shows interchangeable service instances offering an identical statistics service on a single host. To access this service, clients request the Statistics V1.0 interface and specify the nil object UUID. The starting entry for the NSI search is a group corresponding to local servers that offer the service (or a profile that refers to that group).


MAYA system
Statistics-service server instance 1
Statistics V1.0 interface
Report_writer V2.0 interface
MAYA system
Statistics-service server instance 2
Statistics V1.0 interface
Report_writer V2.0 interface
Exporting
Exporting
Name service database
/.:/LandS/Statistics_service_AZTEC
Interface ID for Statistics V1.0
/.:/LandS/Statistics_service_MAYA
Interface ID for Statistics V1.0
/.:/LandS/Statistics_service_grp
/.../C=US/O=TheU/OU=MadCity/LandS/Statistics_service_AZTEC
/.../C=US/O=TheU/OU=MadCity/LandS/Statistics_service_MAYA

Search Requirements
Target Interface: Statistics V1.0
Target Object: None
Starting Entry: /.:/LandS/Statistics_service_grp
Maximum number of traversed entries: 2

Figure 42. Service Model: Interchangeable Instances on One Host

Note that, if an application with interchangeable server instances uses the connectionless RPC protocol, the default behavior of the endpoint map service is to always return the endpoint from the first map element for that set of server instances. To avoid having all clients using only one of the instances, before making a remote procedure call to the server, each client must inquire for an endpoint. For a random selection, a client calls the **rpc_ep_resolve_binding()** routine. Alternatively, a client can call the **rpc_mgmt_ep_elt_inq_...()** routines to obtain all the map elements for compatible server instances, and then use an application-specific selection algorithm to select one of the returned elements.

Distinct Service Instances on a Single Host

With the service model, when multiple server instances on a given host are somehow unique, each instance must export to a separate server entry. The exported binding information must contain one or more instance-specific, well-known endpoints or an instance UUID. Well-known endpoints and instance UUIDs are used under the following circumstances:

- Well-known endpoints

  An instance-specific, well-known endpoint must be provided to a server instance as part of its installation; for example, as a command-line argument. Before calling the export operation, the server instance tells the RPC runtime to use each of its well-known endpoints; it does this by calling **rpc_server_use_protseq_ep()**. The runtime includes these endpoints in the instance's binding information, which the runtime makes available to the instance via a list of server binding handles. The server instance uses this list of binding handles to export its binding information, including the well-known endpoints. The server also uses this list of binding handles to export its well-known endpoint with the local endpoint map; it does this by calling **rpc_ep_register()** or **rpc_ep_register_no_replace()**. Remote calls made using an imported well-known endpoint from a server entry are guaranteed by the RPC runtime to go only to the server instance that exported the endpoint to that entry.

Note: Only one server instance per system can use a well-known endpoint obtained from a given interface specification.

- Instance UUID

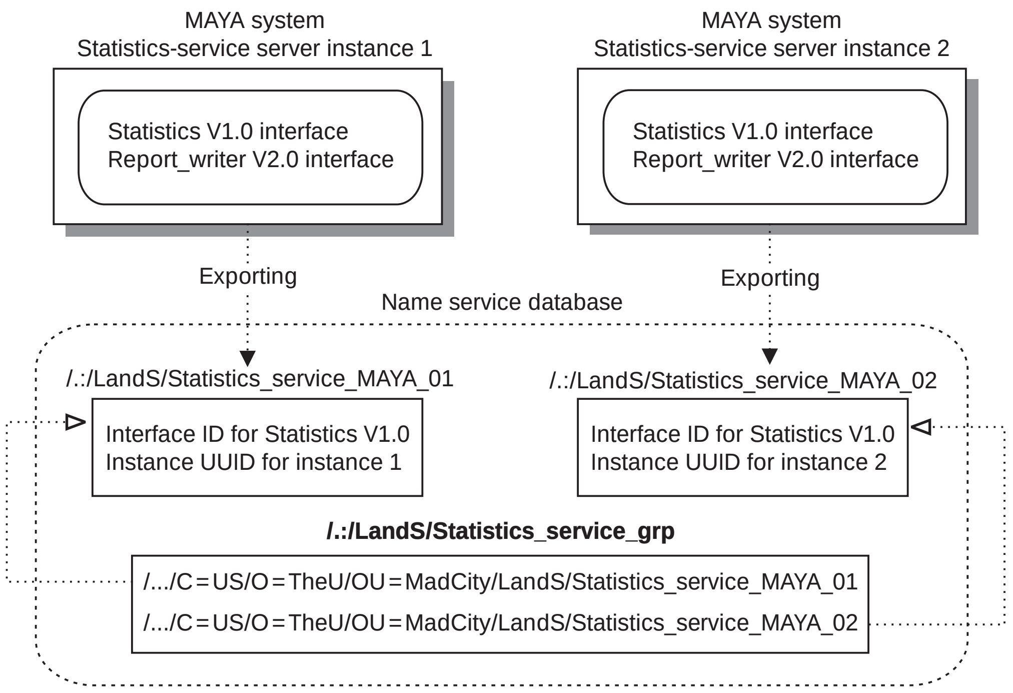
  Create an instance UUID only for a new server entry. Generating a new instance UUID each time a server instance exports to a server entry will result in many instance UUIDs that are difficult to manage and may affect performance as new instance UUIDs are constantly added to server entries. If a new server instance inherits a currently unused server entry left behind by an earlier instance, before exporting, the new server instance should inquire for an instance UUID in the server entry; this is done by calling the **rpc_ns_entry_object_inq_**{**begin**,**next**,**done**}**()** routines. If the inherited entry contains an instance UUID, the server uses it for an instance UUID, rather than creating and exporting a new instance UUID. If an inherited entry lacks an instance UUID, however, the server must create a UUID and export it to the server entry.

  Note that every server instance must register its instance UUID along with its endpoints in the local endpoint map.

  Note: Using an instance UUID precludes any other use of object UUIDs for the application.

Figure 43 on page 233 shows distinct instances of a statistics-service server on the same host. Each server instance uses an instance UUID to identify itself to clients. The instance UUID is the only object UUID a server instance exports to its server entry. Starting at the statistics-service group, clients import the statistics interface.

After finding a server entry with compatible binding information for the statistics interface, the import operation returns an instance UUID along with binding information. Every remote procedure call made with that binding information goes to the server instance that exported the instance UUID.


MAYA system
Statistics-service server instance 1
Statistics V1.0 interface
Report_writer V2.0 interface
MAYA system
Statistics-service server instance 2
Statistics V1.0 interface
Report_writer V2.0 interface
Exporting
Exporting
Name service database
/.:/LandS/Statistics_service_MAYA_01
Interface ID for Statistics V1.0
Instance UUID for instance 1
/.:/LandS/Statistics_service_MAYA_02
Interface ID for Statistics V1.0
Instance UUID for instance 2
/.:/LandS/Statistics_service_grp
/.../C=US/O=TheU/OU=MadCity/LandS/Statistics_service_MAYA_01
/.../C=US/O=TheU/OU=MadCity/LandS/Statistics_service_MAYA_02

Search Requirements
Target Interface: Statistics V1.0
Target Object: None
Starting Entry: /.:/LandS/Statistics_service_grp
Maximum number of traversed entries: 2

Figure 43. Service Model: Distinct Instances on One Host

The Resource Model for Defining Servers

The resource model views servers and clients as manipulating resources. A server and its clients use object UUIDs to identify specific resources. With the resource model, any resource an application's servers and clients manipulate using an object UUID is considered an RPC resource. Typically, an RPC resource is a physical resource such as a database. However, an RPC resource may be abstract; for example, a print format such as ASCII. Note that an application that uses the resource model for one context may use the service model for another. (See earlier sections for details of the service model.)

Applications use object UUIDs to refer to resources as follows:

1. Servers offer resources by assigning an object UUID to each specific resource.
2. Clients obtain those object UUIDs and use them to learn about a server that offers a given resource.
3. When making a remote procedure call, a client requests a resource by passing its UUID as part of the server binding information.

Each RPC resource or type of resource requires its own object UUID. A calendar server, for example, may require a distinct UUID to identify each calendar.

RPC interfaces can be defined to operate with different types of resources and can be implemented separately for each type; for example, a print server application that supports PostScript, sixel, and ASCII file formats. When using different implementations of an interface (different managers), servers must associate the object UUID of a resource, such as an ASCII file format and its manager, by assigning them a single type UUID. To request the resource, a client specifies its object UUID in the server binding information. When a print server receives the remote procedure call, it looks up the corresponding type UUID and selects the associated manager.

Some RPC resources, such as print queues, belong exclusively to a single server instance. Some can be shared among server instances; for example, a file format or an airline reservation database. For server instances on the same system, sharing a resource means that its object UUID cannot distinguish between the two instances. For a print server, this is unlikely to be a problem, assuming that each printer runs only one instance of the print server. In contrast, an application with a widely accessed database, such as an airline reservation application, may need to ensure that clients can distinguish server instances from each other. An application can distinguish itself by supplying its clients with instance-specific information; for example, a well-known endpoint or an instance UUID.

Note: Multiple server instances that access the same set of resources can introduce concurrency control problems, such as two instances accessing a tape drive at the same time. Also, where the system provides concurrency control, servers may compete and have to wait for resources such as databases. Dealing with delayed access to shared resources may require an application-specific mechanism, such as queuing access requests.

Guidelines for Defining and Using RPC Resources

When developing an RPC application, you need to decide whether to use object UUIDs to identify RPC resources and, if so, what sorts of resources receive UUIDs that servers export to the namespace. When making these decisions, consider the following questions:

- Will users need to select a server entry from the namespace based on what object UUIDs the entry contains (and what the client needs)?

  If yes, then a client must specify an object UUID to the import operation.
- Does the type of resource you are using last for a long time (months or years), so you can advertise object UUIDs efficiently in the namespace?

  The information kept in a namespace should be static or rarely change. For example, print queues are appropriate RPC resources. In contrast, quickly changing information, such as the jobs queued for the printer, owners of the jobs, or the time the job was added to the queue, should not be viewed as RPC resources. Such short-lived data may be viewed as local objects, which are stored and managed at a specific server. Programming with local objects is in the area of regular object-oriented programming and is independent of an application’s use of RPC resources.
- Is the number of objects belonging to the type of resource bounded in order to avoid placing high demands on the directory service?
- Will the server implement an interface for different types of a resource, such as different forms of calendar databases or different types of queues?

  If yes, then the server must classify objects into types. For each type, the server generates a nonnil UUID for the type UUID, sets the type UUID for every object of the type, and specifies that type as the manager type when registering the interface. When making a remote procedure call to the interface, a client must supply an object UUID to specify an RPC resource.

- Is control over specific resources an important factor for distinguishing among server instances on a host?

  If yes, then each server must generate an object UUID for each of its resources.

For some applications, such as those accessing a database that many people use, shared access to one or more objects may be essential. However, not all objects accommodate such shared access.

Using Objects and Groups Together

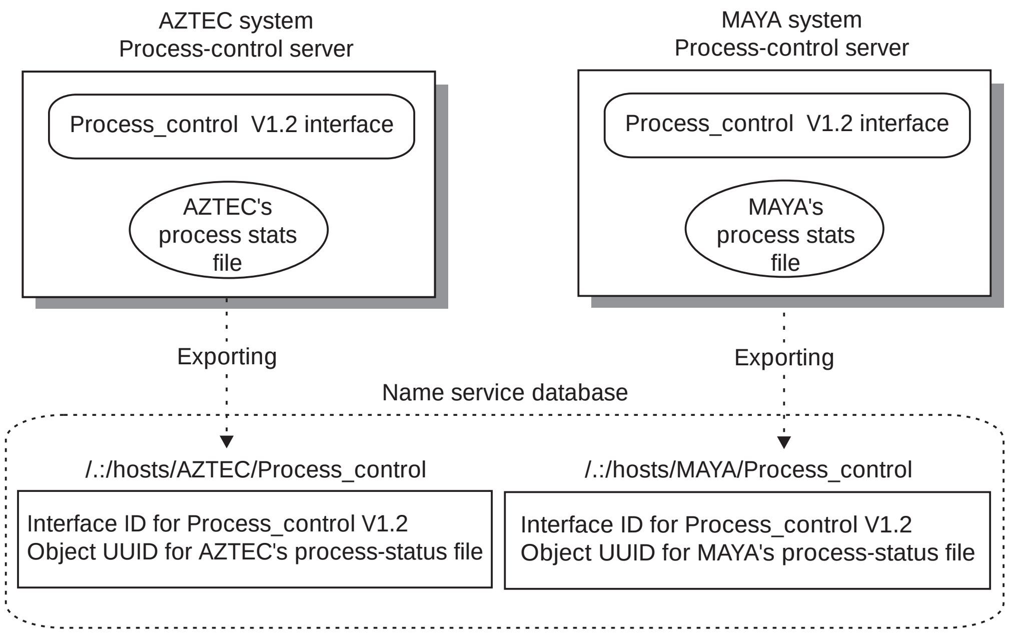
Servers can associate object UUIDs with a group. Each server exports one or more object UUIDs (without exporting any binding information) to the directory service entry of the group. This involves specifying the NULL interface identifier to the export operation along with the list of object UUIDs. The object UUIDs reside in the directory service entry of the group. If a server stops offering an advertised object, a server must unexport its object UUID from the group entry in order to keep its object list up-to-date.

Clients use objects in a group entry as follows:

1. The client inquires for an object UUID from the group entry by calling the **rpc_ns_entry_object_inq_**{**begin**,**next**,**done**}**()** routines. This routine selects one object UUID at random and returns it to the client.
2. The client imports binding information for the returned object UUID (and the interface of the called remote procedure), specifying the group for the start of the search.
3. The import operation returns a binding handle that refers to the requested object UUID and binding information for a server that offers the corresponding object.
4. The client issues the remote procedure call by using that binding handle.
5. The server looks up the type of the requested object.
6. The server assigns the remote procedure call to the manager that implements the called remote procedure for that type of object.

System-Specific Applications

For some applications, the clients need to import an RPC resource that belongs to a specific system, and the clients can specify a server entry name to learn about a server on that system. For example, a process server that allows clients to monitor and control processes on a remote machine is useful only to that machine. Figure 44 on page 236 illustrates this type of system-specific interpretation of the resource model.


AZTEC system
Process-control server
Process_control V1.2 interface
AZTEC's process stats file
MAYA system
Process-control server
Process_control V1.2 interface
MAYA's process stats file
Exporting
Exporting
Name service database
/.:/hosts/AZTEC/Process_control
Interface ID for Process_control V1.2
Object UUID for AZTEC's process-status file
/.:/hosts/MAYA/Process_control
Interface ID for Process_control V1.2
Object UUID for MAYA's process-status file

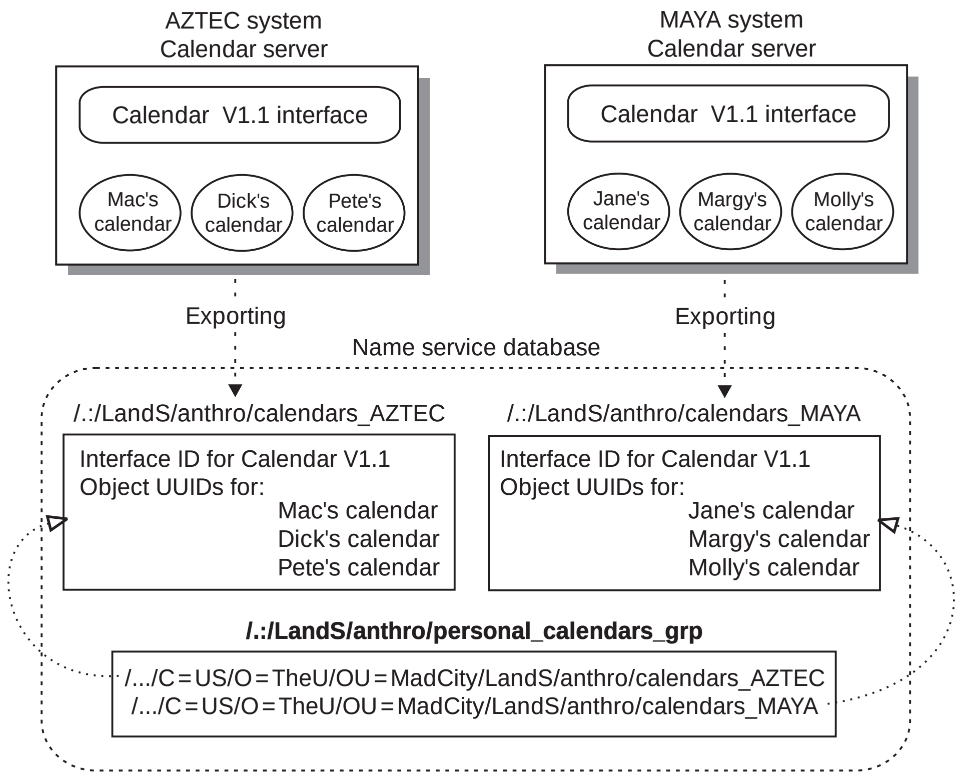
Search Requirements
Target Interface: Process_control V1.2
Target Object: Process-status file of MAYA system
Starting Entry: /.:/hosts/MAYA/Process_control
Maximum number of traversed entries: 1

Figure 44. Resource Model: A System-Specific Application

Because clients usually find a system-specific server by specifying its server entry to the import operation, groups are usually not part of the NSI search path for system-specific applications. However, groups are a management tool for such applications. A group containing the names of the server entries of all the current servers can act as an accounting database. Also, a group for the servers on each set of related systems, such as the members of a LAN or an administrative grouping, permits a client to sequentially use the application on every system in the set. An application with system-specific servers should *not* use profiles.

Exporting Multiple Object UUIDs to a Single Server Entry

Often a single server offers more than one resource, or it offers several types of resources. In cases where a server instance has a large number of object UUIDs, the application should usually place multiple object UUIDs into a single server entry. Typically, an application places all its object UUIDs into one server entry; however, it may need to segregate them into several server entries according to factors such as object type, location, or who uses the different types of objects. When you are subsetting resources, try to assign each resource to a single set so that its object UUID is exported to only one server entry. Figure 45 on page 237 illustrates a single server entry implementation for each server for the resource model.



Search Requirements
Target Interface: Calendar V1.1
Target Object: A specific personal calendar
Starting Entry: /.:/LandS/anthro/personal_calendars_grp
Maximum number of traversed entries: 3

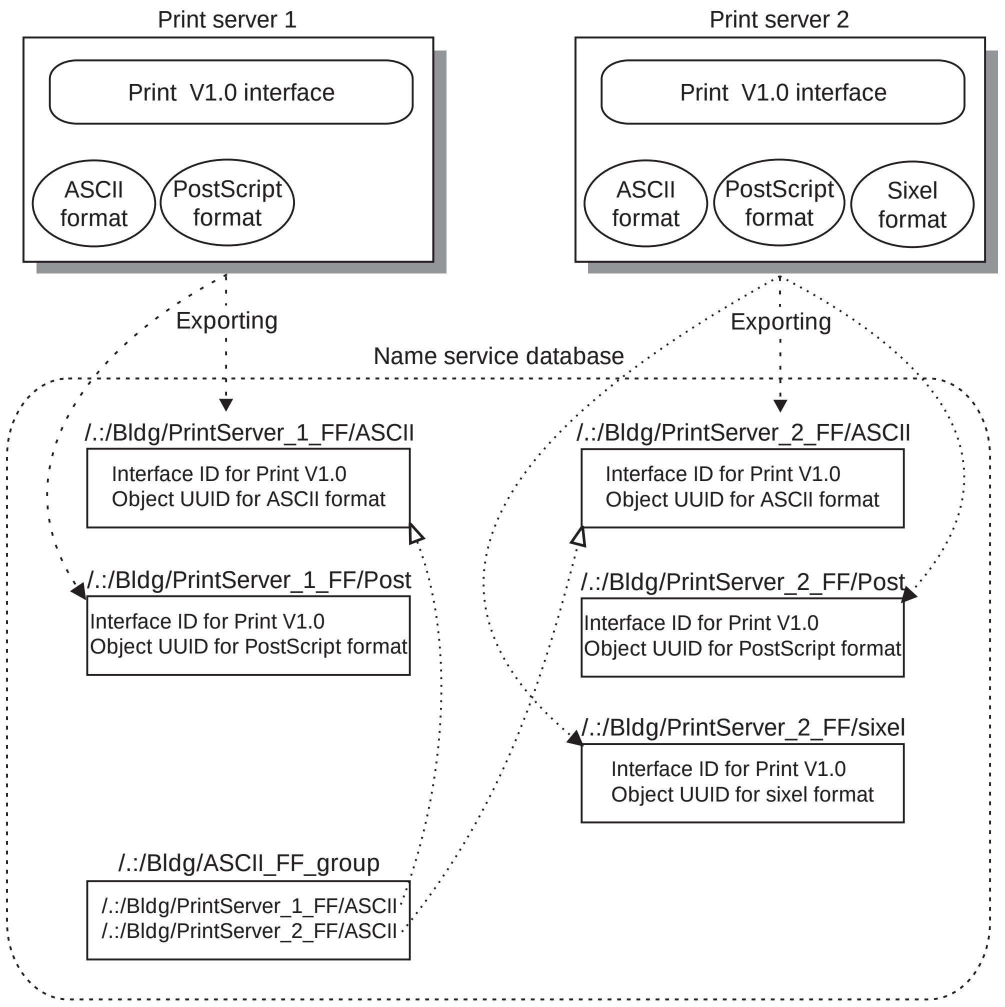
Figure 45. Resource Model: A Single Server Entry for Each Server

Exporting Every Object UUID to a Separate Server Entry

For some applications, exporting each object UUID to a separate server entry is a practical strategy. To avoid excessive demands on directory service resources, however, this strategy requires that the set of objects remain small. Applications with many RPC resources should usually have each server create a single server entry for itself and export the object UUIDs of the resources it offers to that server entry. For example, an application that accesses a different personal calendar for every member of an organization needs to avoid using a separate server entry for each calendar.

For some applications, however, you can use a separate server entry for each object UUID; for example, a print server application that supports a small number of file formats. Each server can create a separate server entry for each supported file format and export its object UUID to that server entry. The server entries for a file format are members of a distinct group.

To import binding information for a server that supports a required file format, a client specifies the nil UUID as the object UUID and the group for that format as the starting entry. The import operation selects a group member at random and goes to the corresponding server entry. Along with binding information, the operation returns the server's object UUID for the requested file format from the server entry. When the client issues a remote procedure call to the server, the imported object UUID correctly identifies the file format the client needs. Figure 46 illustrates this use of object UUIDs.


Print server 1
Print V1.0 interface
ASCII format
PostScript format
Print server 2
Print V1.0 interface
ASCII format
PostScript format
Sixel format
Exporting
Exporting
Name service database
/.:/Bldg/PrintServer_1_FF/ASCII
Interface ID for Print V1.0
Object UUID for ASCII format
/.:/Bldg/PrintServer_1_FF/Post
Interface ID for Print V1.0
Object UUID for PostScript format
/.:/Bldg/PrintServer_2_FF/ASCII
Interface ID for Print V1.0
Object UUID for ASCII format
/.:/Bldg/PrintServer_2_FF/Post
Interface ID for Print V1.0
Object UUID for PostScript format
/.:/Bldg/PrintServer_2_FF/sixel
Interface ID for Print V1.0
Object UUID for sixel format
/.:/Bldg/ASCII_FF_group
/.:/Bldg/PrintServer_1_FF/ASCII
/.:/Bldg/PrintServer_2_FF/ASCII

Search Requirements

Target Interface: Print V1.0

Target Object: ASCII file format (client specifies nil object UUID)

Starting Entry: /.:/Bldg/ASCII_FF_group

Maximum number of traversed entries: 2

Figure 46. Resource Model: A Separate Server Entry for Each Object

Applications that use a separate entry for each object UUID need to use groups cautiously. Keeping groups small when clients are requesting a specific object is essential because an NSI search looks up the group members in random order. Therefore, the members of a group form a localized flat NSI search path rather than the hierarchical path. Flat search paths are inefficient because the average search will look at half the members. Small groups are not a problem. For example, if a group contains only 4 members, each of whom refers to a server entry that advertises a distinct set of RPC resources, the average number of server entries accessed in each search is 2 entries and the maximum is only 4. The larger the group, however, the more inefficient the resulting search path. For example, for a group containing 12 members, each of whom refers to a server entry that advertises a distinct set of object UUIDs, the average search accesses 6 entries and some searches access all 12 server entries.

Chapter 15. Developing Applications that Use Distributed Objects

Before you read this chapter and begin developing with distributed objects, first read *OSF DCE Application Development Guide—Introduction and Style Guide*, chapter 8. This chapter describes how to develop object-oriented, DCE applications that have distributed objects. The chapter introduces C++ features of the Interface Definition Language (IDL) that allow direct development of C++ DCE applications. It covers the following topics:

- IDL and the class hierarchy of a DCE application
- Servers that manage distributed objects
- Clients that use distributed objects
- Multiple interfaces and interface inheritance
- Integrating C and C++ clients and servers
- Using objects from class libraries as RPC parameters

IDL and the Class Hierarchy of a DCE Application

When you develop a DCE application, be it object oriented or otherwise, you begin by creating an interface definition file. This file specifies the operations (with necessary data structures) available for a client to call, all of which a server of this interface must implement.

Although IDL resembles the C programming language, it is intended to be language independent. This means that the applications that are developed could use any programming language that makes sense for them. However, the IDL compiler generates intermediate stub files in either C or C++, two of the most popular languages in use today. We use a particular programming language to take advantage of its features when developing an application. However, an application developed in another language could use mechanisms (such as wrapper routines) that call the routines generated by the IDL compiler.

The **-lang** option of the IDL compiler when used with a **cxx** argument generates C++ intermediate stub files rather than C intermediate stub files. In order to support the generation of C++ stubs, the IDL also needs additional features to give applications developed in C++ a cleaner and more efficient use of the distributed application features of DCE. This section describes how to use these features in interface definitions.

Specifying a C++ Class via an IDL Interface

An IDL interface definition and a C++ class are very similar. An IDL interface definition specifies the data structures and operations that an application needs to use a distributed interface. A C++ class specifies the data structures and functions that an application needs to use a type of object. IDL and the IDL compiler blend together the distributed computing capabilities of an interface definition with the object-oriented features of a C++ class to specify *distributed objects*.

The following example shows an interface definition for a distributed Matrix object:

```
FILE matrix.idl
---------------------------------------------------
```

```
[
  uuid(24cb0eda-3eb9-11ce-b1ce-08002bbbf636)
] interface Matrix
{
  /* Create a new 2 by 2 Matrix. This operation requires an ACF */
  /* to tell the stubs this is a creator operation. */
  Matrix * createMatrix(
[in] long v11,
[in] long v12,
[in] long v21,
[in] long v22
  );

  /* Create a new Matrix of size rows by columns and return TRUE.*/
  /* If server does not support the size requested, return FALSE */
  /* and the maximum size in rows and columns that it supports.*/
  /* This operation requires an ACF to tell the stubs this is a  */
  /* static operation. */
  boolean newMatrix(
[in, out] long &rows,
[in, out] long &columns,
[out] Matrix ** m
  );

  /* The rest of the operations operate on the existing object */
  /* that invokes them (this Matrix). */

  /* Set a new value in this Matrix. */
  void set(
[in] long row,
[in] long col,
[in] long value
  );

  /* Get a value from this Matrix. */
  long get(
[in] long row,
[in] long col
  );

  /* Return a new Matrix that is the inverse of this Matrix. */
  Matrix * inverse();

  /* Return a new Matrix that is the product of this Matrix */
  /* and m1.*/
  Matrix * multiply(
[in] Matrix * m1
  );

  /* Return in Matrix m2 the sum of this Matrix and m1. */
  void add(
[in] Matrix * m1,
[out] Matrix ** m2
  );

}
```

Interface definitions and C++ classes are both specifications, not implementations. An implementation of the IDL interface definition is a server's manager code, and an instance of the class is an object of that class. The operations of the Matrix interface are described as follows:

createMatrix

If an interface designer expects clients to create dynamic objects, at least one operation must be a static function that creates a new object. For example, the **createMatrix** operation is intended to be a static member function. Static member functions do not require an existing object before they are called.

newMatrix
: Parameters are typically passed-by-value in C++. Apply the reference operator (**&**) to parameters you want to pass by reference. A reference parameter is required if the function changes the value. In this example, the number of the rows and columns are input but the values change if the function cannot create the Matrix requested.

set
: The **set** operation sets an individual value in an existing Matrix object.

get
: The **get** operation obtains an individual value from an existing Matrix object.

inverse
: The **inverse** operation returns a new Matrix object that is the inverse of an existing Matrix object.

add
: The **add** operation does not return a value but has an output parameter that is a new Matrix object. The output is the sum of an existing Matrix object and an input parameter that is a Matrix object.

It is not appropriate to encapsulate data in the definition of a public interface; that would be implementation detail, which does not belong in the interface definition. Therefore, there is no IDL concept of *private* data as there is in C++. However, the base class from which all IDL interface classes are derived does encapsulated binding data and RPC mechanisms.

IDL-Generated Classes as Part of Your Hierarchy

The interface definition is compiled with the following IDL compiler command to generate an intermediate C++ header file, client stub, server stub, and manager class header file:

```
idl -lang cxx matrix.idl
```

The IDL compiler then automatically invokes the local language compiler by default. In this case it invokes the C++ compiler to create binary stub files that are used in the development of clients and servers.

The IDL compiler automatically uses an Attribute Configuration File (ACF) if one is available in its search directories. C++ applications use an ACF to specify features such as static member functions, implementation class names, a lookup function for named or persistent objects, and header files for inclusion in stub files.

Class hierarchies are created by the IDL compiler and made part of the clients and servers. The hierarchies include an RPC base class that encapsulates the distributed nature of objects. An abstract interface class is also created by the IDL compiler and derived from the RPC base class. The interface class includes the data types and nonstatic member functions of the interface definition and is the common interface used by clients and servers. The interface class contains no implementation. Therefore, clients and servers each need to derive implementation classes from the interface class. Clients have an **idl**-generated implementation class generated for them, called a *proxy class*. Servers have an **idl**-generated class generated for them called a *manager class*.

Server developers must implement the manager class functions by either modifying the generated manager class header file or deriving an application-specific class from the manager class. The manager class name generated by the IDL compiler is a combination of the interface name and **Mgr**. For example, the manager class for our **Matrix** interface is **MatrixMgr**. The generated class is placed in a header file with a name created from the combination the interface file name and the **_mgr** suffix. For example, **matrix_mgr.h**.

If you decide to implement the manager by modify the generated manager class, you want to be sure that any subsequent invocation of the IDL compiler does not overwrite your manager class code. Use the -no_cxxmgr option with the IDL compiler command to suppress generating a manager class.

```
idl -lang cxx -no_cxxmgr matrix.idl
```

Servers that Manage Distributed Objects

An application creates local objects for its own internal use. Servers must manage these application-specific objects, just as any application does. In addition, distributed object servers must manage two other basic kinds of objects for their clients: distributed dynamic objects and distributed named objects. The following subsections cover a number of programming tasks and topics of interest to server developers. The basic programming tasks server developers typically perform include the following:

1. Implementing dynamic objects for clients, if needed
2. Implementing static member functions, if any
3. If clients can pass local objects in calls to the server, linking in both server and client stubs so that manager functions automatically access parameters that are client-local objects

Other topics of interest to server developers include naming objects, dynamically creating named or persistent objects with a lookup function, and using the DCE backing store for persistent objects.

Initializing Object-Oriented Servers

DCE servers consist of two major portions of code: initialization code and manager code. All servers must perform some initialization prior to providing services and objects. In addition to initialization code, the server also has manager code that implements each interface that a server supports. The manager code contains the implementation of both the static and nonstatic member functions (or methods).

In an object-oriented development environment, there would generally be a server class with each server being an instance of that class. Although interesting and important for object-oriented applications, the design and implementation of such an environment is beyond the scope of this chapter. However, there are a few issues to consider when initializing C++ servers, as follows:

- Server Registration

  Servers are automatically registered by server stubs, so if your code calls the **rpc_server_register_if()** routine, you will get a warning indicating the server is already registered.
- Entry point vectors (EPVs) and C++ function tables

When a C++ application is compiled, a function table is automatically generated for each class. The EPV mechanism in DCE is necessary for languages that do not supply such a feature, such as C. Use NULL (the default EPV) for C++ applications.

- Named and persistent objects

  Your server may need to create persistent or long-lived named objects before the server begins servicing client requests. Naming objects is described later in this chapter.
- Exceptions

  DCE supplies exception handling macros such as **TRY,CATCH**, and **FINALLY** for use in distributed applications. You should use DCE's macros in your applications instead of the standard C++ macros to be sure exceptions are propagated properly from servers to clients.

See the first chapter of the *OSF DCE Application Development Guide—Introduction and Style Guide* for the typical steps DCE requires to initialize a server. The following subsections describe features needed in manager code, the code responsible for a server's specific implementations of the interfaces supported.

Implementing Distributed-Dynamic Objects

After the server has been initialized and is listening for calls, one obvious question arises: How does the server create distributed objects? The server creates objects locally, just as they are created in typical C++ applications, by allocating variables of the class types or by dynamically creating them with the C++ **new** operator, as shown in the following example:

```
// m1 is allocated as a variable of class Matrix
Matrix m1;
// m2 is a pointer variable of class Matrix allocated with "new"
Matrix *m2 = new MatrixMgr(0, 0, 0, 0);
```

However, clients have no way to use these objects since they are only local and not yet available as distributed objects.

For distributed dynamic objects, the server needs a way to know when a client requests that the server create a dynamic object. This is done by using an ACF to associate an appropriate interface operation with the server's implemented manager class. You must then write manager code that turns server-local objects into distributed dynamic objects.

When you compile the interface definition file to create the interface header file and server stub, you use an ACF to customize how your application code uses the interface, as shown in the following:

```
/* FILE NAME: matrix.acf */
/* This file defines some attributes for the Matrix interface */
interface Matrix
{
/* include header files generated into the server stub */
[sstub] include "matrix_mgr";

/* createMatrix should be mapped as a creator function.    */
/* The MatrixMgr is a class derived from the interface class. */
```

```
[cxx_new(MatrixMgr)] createMatrix();
.
.
.
```

[sstub] include

Use the **include** statement with the**sstub** attribute to make the IDL compiler include specific header files in the server stub. In this example, this is required so that the stub has a declaration of the manager class.

[cxx_new(MatrixMgr)] createMatrix();

Use the **cxx_new** attribute with the name of the implemented manager class (**MatrixMgr**) as an argument, and apply it to the interface operation that is intended to create a dynamic object, **createMatrix.** The manager class can be the **idl**-generated one, as in this example, or it can be one you derived from the generated manager class.

The following C++ code shows examples of constructor and destructor functions you write for the manager class (**MatrixMgr**):

```
.
.
.
// Constructor
MatrixMgr::MatrixMgr(idl_long_int v1, idl_long_int v2,
           idl_long_int v3, idl_long_int v4)
{
d[0][0] = v1;
d[0][1] = v2;
d[1][0] = v3;
d[1][1] = v4;
}

// Destructor for a 2x2 Matrix.
// In this application, the destructor does nothing.
MatrixMgr::
˜ MatrixMgr(void)
{
return;
}
```

MatrixMgr::MatrixMgr

In the trivial case, a constructor automatically initializes the object allocated by the C++ new operator. For this application, the constructor simply fills in the data structure with the values sent in the remote procedure call. In more realistic applications, the C++ constructor may have to perform additional work. C++ allows you to define constructor functions that contain application-specific code that is automatically called immediately after the object is created.

MatrixMgr:: ˜MatrixMgr

In addition to a constructor, C++ allows you to define destructor code that is called to do application-specific cleanup just prior to the release of storage for the object. In this example, the destructor is a dummy function that has no special code and does nothing.

When a client initiates the creation of a dynamic object, the server receives a remote procedure call request for the **createMatrix** function. This causes the server stub to call the C++ constructor for the specified manager class (in this example **MatrixMgr**), which creates a new object on the server. When this happens, the

DCE runtime stores information about the object in a table that also associates the object with the requesting client. No other clients have access to a dynamic object unless the originating client gives an object reference to another client. The runtime uses reference counting to keep track of how many clients know about the object. When a client deletes the object, the reference count on the server is reduced. The object on the server is deleted only when the reference count reaches zero, which indicates that there are no more clients with references to the object.

Implementing Static Member Functions

Static member functions are specified in the interface definition or with an ACF. Those operations of an interface that the designer knows should be static have the **static** keyword before the operation in the interface definition file. For example, the **newMatrix** operation of the Matrix interface is designed to work without an invoking Matrix, so it could have been specified in the interface definition as a static member function as follows:

```
interface Matrix
.
.
.
static boolean newMatrix(...);
```

The IDL compiler automatically compiles this kind of operation as a static member function in both the server and client stubs. Depending on how a developer wants to implement the interface, it may be undesirable to commit to a static function. For example, the **CreateMatrix()** function described in the previous section could have been specified as static in the interface, but it would prevent the server developer from directly using the built-in constructor feature of C++ to implement an object creator function. Therefore, to give maximum flexibility to both client and server developers, the static keyword can be left off the operation and then specified as needed in an ACF file.

Of course, creating new objects is just one thing a static member function can do, and so any number of other static member functions may be specified in the interface to do whatever application-specific work is required.

The Matrix interface declares the following operations:

```
  Matrix * createMatrix(
[in] long v11,
[in] long v12,
[in] long v21,
[in] long v22
  );

  boolean newMatrix(
[in, out] long  &rows,
[in, out] long  &columns,
[out]   Matrix ** m
  );
```

The IDL compiler requires an ACF to implement these as static member functions. A sample server ACF contains the following:

```
/* FILE NAME: matrix.acf */
/* This file defines some attributes for the Matrix interface */

interface Matrix
```

```
{
/* include header files generated into the server stub */
[sstub] include "matrix_mgr";

/* createMatrix should be mapped as a creator function.    */
/* The MatrixMgr is a class derived from the interface class. */
[cxx_new(MatrixMgr)] createMatrix();

   /* newMatrix should be mapped as a static member function. */
[cxx_static] newMatrix();
}
```

[sstub] include

Use the **include** statement with the **sstub** attribute to make the IDL compiler include specific header files in the server stub. In this example, this is required so that the stub has a declaration of the manager class.

[cxx_new(MatrixMgr)] createMatrix();

Use the **cxx_new** attribute with the name of the implemented manager class (**MatrixMgr**) as an argument, and apply it to the interface operation that is intended to create a dynamic object, **createMatrix**. This feature is described in the previous section.

[cxx_static] newMatrix;

Apply the **cxx_static** attribute to the names of all interface operations you intend to implement as static member functions.

To complete the story of a static function, the following is an example of one trivial implementation of the **newMatrix** function. The code implements only a 2 by 2 Matrix. If a client inputs values other than **2** for **rows** or **columns**, the values are changed to **2**, and **FALSE** is returned.

```
// Implementation of the static member function declared with an ACF
idl_boolean
Matrix::newMatrix(idl_long_int &rows, idl_long_int &columns, Matrix **m)
{
if(rows != 2 && columns != 2) //implementing only a 2 by 2 Matrix
{
  rows = columns = 2;
  *m = 0;
  return FALSE;
}
else
{
  *m = new MatrixMgr(0, 0, 0, 0);
  return TRUE;
}
}
```

The **cxx_static** attribute can also take an argument that represents a new name to use for the function. This may be necessary in your application if it needs to distinguish between remote and local versions of a static member function. In any case, to minimize changes to your code modules, it is a good idea to keep the implementation of static functions in files separate from the nonstatic member functions (the rest of the manager code). The following section describes a common example of when to use an argument for the **cxx_static** attribute.

When Function Parameters Are Remote Objects

With distributed applications, especially with distributed objects, the distinction between a client and server is not determined so much by a program on a specific machine as it is by a state the program is currently in. Thus a program can be both a client and a server, depending on its purposes. For example, it is possible that, when a client object uses a function that has another object as an input parameter (our **add()** function, for example), the input could easily be an object that is local to the client. When the server is executing the **add()** function, it needs a way to transparently access the input object that is now remote to the server (but local to the client).

A server accesses client objects by linking in the client stub in addition to the server stub. When an object is remote to the server, information in the binding from a client is used automatically by the server stub during unmarshalling to create an object reference (a proxy) and make remote calls back to the client to access the object there. Server code itself does not have to do any special calls.

The following figure illustrates a brief review of all the code modules a typical server needs. The server stub for each interface and initialization code are required to access DCE's distributed environment. Each interface requires a manager class, manager code, and static member functions to implement them. Each interface with input object parameters should include the client stub in order for the server stub to access client-local objects.

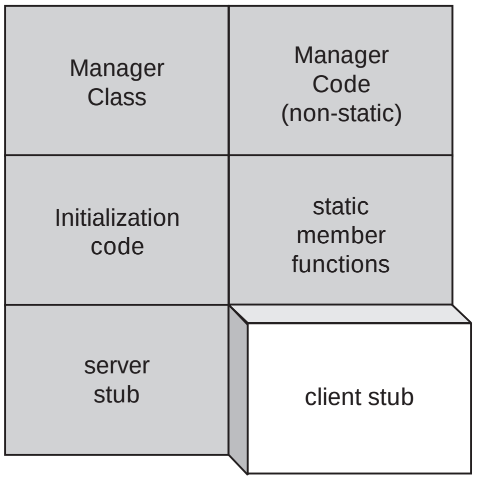


Figure 47. Servers Need the Client Stub to Access Client-Local Objects

Making this work includes one other step besides linking in a client stub. If there are any static member functions in the interface class, linking together a client and server stub will produce a C++ compiler error due to a name conflict in the server and client stub versions of the function. You use the **cxx_static** attribute in an ACF to rename the server's local version of the static functions. An example of such an ACF is as follows:

```
/* FILE NAME: matrix.acf */

interface Matrix
{
/* include files generated into the server stub */
[sstub] include "matrix_mgr", "staticfunc";
/* createMatrix should be mapped as a creator function. The */
/* argument represents the class that implements the interface.*/
[cxx_new(MatrixMgr)] createMatrix();

/* newMatrix should be mapped as a static member function. */
[cxx_static(LocalMatrix)] newMatrix();
}
```

The **LocalMatrix** argument to the **cxx_static** attribute is the name for this server's local implementation of the function, and the **newMatrix** name refers to the remote (in this case, the client stub) function.

The **include** statement is needed with the **sstub** attribute to include header files that contain declarations needed by the server stub only. In this example, the **matrix_mgr.h** file contains the server's manager class declaration, and the **staticfunc.h** file contains the declaration of the renamed static function, **LocalMatrix**. The **staticfunc.h** header file is as follows:

```
// FILE NAME: staticfunc.h
// This file declares the function(s) to call
// when invoking local versions of an interface's static functions.
// The prototype signatures should match that of the remote versions.

idl_boolean LocalMatrix(idl_long_int, idl_long_int, Matrix **m);
```

Note: Servers will work without including the client stub and renaming the local version of the static functions. However, if a client ever uses a member function with an object parameter that is remote to the server, a runtime error occurs. If this happens, the server raises an exception (**rpc_x_no_client_stub**) to propagate back to the client that indicates the client stub is not included in the server.

Naming Objects

This section explains how to do the following for named objects:

- Register named objects

  The built-in **register_named_object()** function uses the name service and endpoint map to name and advertise an object.
- Place object names in name service directly

  The name service can be used to advertise objects for which an instance has not yet been created.
- Dynamically create instances of named or persistent objects

  The server's runtime can automatically call a lookup function to create objects it supports but does not yet have an instance created.

Registering Named Objects

DCE RPC supplies every interface class with a member function, **register_named_object()**, to do all that is required to register named objects. The following example shows how a server might create an object and then register it as a named object. The server first creates an object by using the **new** operator. Then the server calls the **register_named_object** function to register the object's name, universal unique identifier (UUID), and its server binding information with the name service. The function also registers the object's UUID and binding information with the host's endpoint map, and it updates the runtime's object table.

```
 .
 .
 .
 // Create an object on the server.
Matrix * matrix = new MatrixMgr(1, 1, 1, 1);
 matrix->register_named_object((unsigned_char_t *) \
     "/.:/MatrixObject");
 .
 .
 .
```

The **register_named_object()** function greatly simplifies your work, but you need to be aware of the information it uses and generates. The following lists the approximate order of events that occur when an object invokes this function:

1. A name service entry is created if one is not already there, using the name in the first argument of the function.
2. If the named object does not already have a UUID associated with it in the name service, one is created.
3. The server's binding information is associated with the name service entry.
4. All interfaces supported by the object are also registered with the name service.
5. The object UUID is associated with the server's location on the host by registering endpoints in the host's endpoint map.

   The function has an optional second argument of type **boolean**. The default value is **TRUE**, which means this is the *only* server on this host that services this interface. (In C++, using no argument is the same as using the default value for the argument.) If the default value is used, values in the endpoint map are updated.

   If the second argument to the function is **FALSE**, this is not the only server this host has that services this interface. In this case, the **register_named_object()** function adds server binding information to the endpoint map (rather than updating the endpoint map) so clients can find any of the servers. See the **rpc_ep_register_no_replace(3rpc)** reference page for more on this topic.
6. Finally, an object table maintained by the server's RPC runtime is updated so that requests for specific objects are directed to the correct member function invocation. Even though creating the object in the first place registers it with the runtime's object table, some information (such as the object's UUID) may need to be updated.

Placing an Object's Name Directly in the Name Service

Consider the following situations:

- Suppose you want a server to just advertise the named objects it supports and not use resources to create them until they are needed. As described in the previous section, an object must be created before it can register itself with the **register_named_object()** function.
- Suppose you want to use a known UUID to represent a named object. If the name does not already exist in the name service, a new UUID is generated via the **register_named_object()** call on the fly. This may be fine for many applications, but, for some, manipulating objects only by their name service names may be cumbersome and inflexible.

If you wish to place a named object in the name service and at the same time use a consistent, stable, and well-known UUID for a named object, you first associate the UUID with the named object in the name service *prior to* using the **register_named_object()** function. There are typically two ways to place object names in the name service:

- Prior to server startup, you can create or update a named object entry by using **dcecp** with the **rpcentry** object and its export operation.
- Your server can create or update a named object entry by using **rpc_ns_binding_export()** during server initialization.

The following example shows a script of **dcecp** commands and arguments to execute on the server's host to export an object's name and then show the data exported to the entry:

```
dcecp -c rpcentry export /.:/objects/IdentityMatrix
-interface {24cb0eba-3eb9-11ce-b1ce-08002bbbf636  0.0} \
-binding  {ncacn_ip_tcp 'hostname'} \
-binding  {ncadg_ip_udp 'hostname'} \
-object  {dcea4900-65ba-11cd-bb34-08002b3d8412}
dcecp -c rpcentry show /.:/objects/IdentityMatrix
```

Attributes and arguments are as follows:

/.:/objects/IdentityMatrix
: The object name.

-interface ...
: The interface's UUID and version numbers from the interface definition header.

-binding *ncacn_ip_tcp* '**hostname**' \
: Binding information including a protocol sequence and the host's name (generated with the **hostname** command).

-object ...
: The object UUID desired.

Whether you call **dcecp**'s **rpcentry export** operation or the **rpc_ns_binding_export()** routine, the first call automatically creates the entry in the name service and each additional call adds binding information to the entry.

Dynamically Creating Named or Persistent Objects

If there are potentially thousands of persistent objects, you may want your application to conserve resources and not register all of them at server startup. Servers may defer creating objects until a client makes a request to use one.

The server's runtime maintains a table of all its objects. The server gets a request from a client on an object that is uniquely identified by an object UUID (from the binding handle), and the object table maps object UUIDs to each object's address in the server's address space. The server's object table is a C++ class containing the following information:

- Object UUIDs
- Interface UUIDs
- Object addresses

If the runtime cannot find the UUID of the object requested, an exception is raised on the server to propagate to the client unless a user-defined *object lookup function* exists. If the lookup function does exist, the runtime automatically executes it. The lookup function is created by the server developer to create the object and, if required, register it as a named object. If the lookup function cannot create an object for the specified UUID, it should return a 0, which causes the runtime to raise an exception (**rpc_x_object_not_found**).

After the named object is registered, the object table contains the new object UUID, so subsequent attempts to use the object do not invoke the lookup function again. Alternatively, the lookup function can maintain its own object map. By not registering with the runtime, subsequent operations will invoke the lookup function. This allows the developer to use the lookup function to maintain complete control over the existence of the object.

A lookup function name is specified using an ACF when the interface is compiled. The following example is a portion of an ACF that specifies a lookup function:

```
[
cxx_lookup(object_lookup)
]
interface Matrix
{
[sstub] include "matrix_mgr", "lookup";
.
.
.
```

cxx_lookup(object_lookup)

To specify a lookup function use the **cxx_lookup** attribute with the name of the lookup function (in this case, **object_lookup**) as an argument. A lookup function is interface-wide, so it is defined in the ACF header.

[sstub] include

Use the include statement with the **sstub** attribute to make the IDL compiler include implementation-specific header files in the server stub. The **matrix_mgr.h** header file contains the manager class and the **lookup.h** file contains a declaration of the lookup function you create.

The following example shows the declaration of the **object_lookup** function in the **lookup.h** header file:

```
//FILE NAME: lookup.h
//This file declares the lookup function used
// for server management of object lookup.

Matrix *object_lookup(uuid_t *);
```

A lookup function has the following signature requirements:

- The lookup function returns a pointer to the interface class (**Matrix ***).
- The function name matches the one declared in the ACF (**object_lookup**).
- There is one input parameter pointer of type **uuid_t**.

An implementation of the **object_lookup()** function is shown in the following section.

Storing and Retrieving Persistent Objects

DCE provides a convenient database storage facility called the backing store, that lets you store and retrieve objects in a system-independent manner. The following implementation of a lookup function shows how to use a backing store database to lookup an object.

```
// FILE NAME: lookup.cxx
// This file contains the server lookup callout function
// specified by the [cxx_lookup] attribute in an ACF. It is called
// whenever an object cannot be found within the DCE runtime.

extern "C" {
#include <dce/dce.h>  // standard DCE header file
#include <dce/dbif.h> // backing store facility header file
}
#include "matrix_mgr.h"
#include <check_status.h>
#include "backing.h" // IDL generated header file
//
// This function performs the server management of object lookups.
// If the uuid_t parameter identifies one of the persistent objects,
// this function creates and returns the object.
//


Matrix *
object_lookup(uuid_t *key)
{
  dce_db_handle_t db_h;
  backing_data_s_t data;
  int  found;
  unsigned32 status;
  Matrix *matrix;

  //
  // Lookup the UUID's in a backing store database
  // and get the data.
  //
  dce_db_open(
   "backing.store",
    0,
    db_c_index_by_uuid | db_c_readonly,
    (dce_db_convert_func_t) backing_data_convert,
    &db_h,
    &status
    );

  dce_db_fetch_by_uuid(
    db_h,
    key,
    (void *) &data,
    &status
    );
  if (status != rpc_s_ok)
    found = 0;
  else
    found = 1;

  dce_db_close(&db_h, &status);

  if (!found)
    return 0;

  // Found the object's data so create an instance of it.
 matrix = new MatrixMgr(data.v00, data.v01, data.v10, data.v11);

  // register the object so clients can find it directly
  // and the server won't have to look it up again.
  matrix->register_named_object((unsigned_char_t *) data.name);

  return matrix;
}
```

The example is described as follows:

dce_db_open()

Applications that have persistent objects commonly store in a database the information necessary to regenerate the object. In this example, the data is stored in a backing store data file (**backing.store**). The argument **db_c_index_by_uuid | db_c_readonly** indicates the file is opened for read-only access and to be indexed by UUID. The argument **backing_data_convert** is the function (defined in the backing interface) that the backing store facility uses to store or retrieve the data. A record for this database is also defined in the backing interface to contain the entry name and values for a two-by-two matrix. The backing interface is in the file **backing.idl**, and looks as follows:

```
[
uuid(3e9400dc-0895-11cf-abec-08002b39f4b8)
] interface backing
{
    import "dce/database.idl";

    /* Data: object name (for CDS) and values      */
    /* for 2-by-2 matrix                 */
    typedef struct backing_data_s_t {
        [string] char name[100];
        int v00;
        int v01;
        int v10;
        int v11;
    } backing_data_s_t;

    /* conversion function declaration */
    void backing_data_convert(
        [in] handle_t            h,
        [in,out] backing_data_s_t    *data,
        [in,out] error_status_t       *st
    );
}
```

dce_db_fetch_by_uuid()

This DCE routine obtains the data for the object represented by the key UUID.

matrix = new MatrixMgr(data.v00, data.v01, data.v10, data.v11);
An instance of the persistent object is created for this server. Note that each server of the object would have its own implementation and instance of the object.

register_named_object((unsigned_char_t *) data.name);

This function registers the object's name, UUID, and its server binding information with the cell's name service. It also registers the object's UUID and binding information with the host's endpoint map. Finally, this routine updates the runtime object table. As an option for more specific control, you can choose not to call this function and implement your own object table instead.

return(0);

If the object was not found, return a 0 value. This will cause the server to raise an exception (**rpc_x_object_not_found**).

The **backing.idl** file is compiled with the following ACF:

```
interface backing
{
    [encode,decode] backing_data_convert(
        [comm_status] st
    );
}
```

The application must also declare the database conversion function as shown in the following **backing.cxx** file:

```
#include "backing.h"

extern "C" {

void
backing_data_convert(
    idl_es_handle_t        h,
    backing_data_s_t    *data
) {
}

}
```

For more on how to use the backing store facility, see the chapter, *The DCE Backing Store*.

Clients That Use Distributed Objects

This section describes how to write object-oriented DCE clients. The subsections describe how to do the following:

- Create remote, dynamic objects
- Create both local and remote instances of the same class
- Call functions that intermix the use of local and remote objects
- Bind to named objects by using names stored in the name service
- Bind to named objects by using their UUID identifiers. This method uses your local name service hierarchy to begin the namespace search.
- Bypass the name service to bind to objects by using binding information

Creating Remote-Dynamic Objects

In C++ we create new objects dynamically by calling the **new** operator for the class. This works to creates local objects, but how do clients create remote dynamic objects? In order for a client to create dynamic objects, at least one static object creator operation must be defined in an interface to create its objects. Remember that a static member function does not have to be invoked by an existing object, and thus it is appropriate as a way to create new objects. The operation can be declared to return a new object as a return value or an output parameter. Object creator functions are declared as static in either of two ways:

- In the IDL file explicitly by using the **static** keyword
- In an ACF by using the **cxx_static** (or **cxx_new**) attribute

The following example is an ACF that specifies object creator member functions for the Matrix interface:

```
/* FILE NAME: matrix.acf       */
/* This file defines some attributes for a simple client  */
/* of the Matrix interface. */
```

```
interface Matrix
{
    /* createMatrix needs to be mapped as a creator function  */
    [cxx_static] createMatrix();

    /* newMatrix needs to be mapped as a static member function */
    [cxx_static] newMatrix();
}
```

When the interface is compiled with this ACF, the IDL compiler generates a proxy class for the client in which these operations are declared as static member functions. The proxy class is our client's interface to DCE, and it is DCE's mechanisms that let clients interact with objects in the distributed environment.

In some ACFs, static object creator functions may be specified with the **cxx_new** attribute instead of **cxx_static**, and both the attributes may include an argument. These differences do not affect the client stub and are significant only for server stubs.

The following example shows how a client calls the Matrix interface's static member functions:

```
#include "matrix.h"        // IDL generated header file
#include printmatrix.h

void
main()
{
    Matrix    *m1;

    cout << "Creating dynamic objects:" << endl;

    // Create a remote Matrix object on a server using an
    // object creator function.
    m1 = Matrix::createMatrix(1, 2, 3, 4);
    cout << "m1 created by an object creator function:" << endl;
    print(m1);
    delete m1;

    // Create a remote Matrix object on a server using
    // a static member function.
    idl_boolean result = Matrix::newMatrix(2, 2, &m1);
    if(result)
    {
      print(m1);
      delete m1;
    }
    .
    .
    .
```

#include ″matrix.h″

The interface class and proxy class are defined in the **matrix.h** header file generated by the IDL compiler.

Matrix *m1;

Object references are declared as pointers to an interface class. The interface class is an abstract class, and C++ does not allow you to create instances of it. However, pointers to abstract classes are allowed. When a remote object is created for one of these object references, the client stub actually creates a proxy class object on the client.

m1 = Matrix::createMatrix(...)

Object creator functions are invoked using the interface class name (Matrix) with the standard C++ scope operator (::). This function creates a remote Matrix object on a server and returns a reference to the remote object.

print(m1);
After a dynamic object is created, the application uses it just like any local object. This function is an application-specific inline function to display a Matrix. It is defined in the **printmatrix.h** header file and uses the Matrix interface's **get()** function.

delete m1;

Remote dynamic objects are deleted with the standard C++ **delete** operator, just like local objects. However, for the remote object, an RPC is sent to it to decrement the reference count. If no other clients have a reference to it, the object is also deleted from the server's address space. Client applications should take care to delete all dynamic objects prior to exiting. Otherwise, the object remains in the server's address space wasting resources. Dynamic objects are created for the use of the invoking client. This means that servers cannot give a different client a reference to a dynamic object. However, the client could behave as a server and give a copy of the object reference to another client. For this reason, a reference count is maintained on the server for objects.

... = Matrix::newMatrix(2, 2, &m1)

Static member functions are invoked using the interface class name (Matrix) with the standard C++ scope operator (**::**). This function is also an object creator function that creates a remote Matrix object on a server.

Creating Client-Local Objects

The client code in the previous section showed only the case in which a class of objects is remote. However, many client applications also need to create and use local objects of the same class. The significant difference is that the local object is not created by way of a remote procedure call as is the remote object. The interface class generated by the IDL compiler from the interface definition is an abstract class. This means that another class must be derived from it to create and manipulate objects. The client stub has a proxy class automatically defined for remote objects, but your client application must define a local implementation class so that your client can create and manipulate local interface objects.

Do the following to create and use local versions of interface objects:

1. Derive a local class from the interface class to implement the client-local objects. This class is just like a manager class used in server development: in fact, this example uses the same manager class as the server.
2. Write the local code that implements the interface class. Our example uses the same manager code implementation as for the server. You implement the manager class by adding the code to the manager header file generated by the IDL compiler, or by deriving a new class from the manager class and implementing those functions.
3. Link the local class and local implementation code into your client application.

The following sample code shows how a client creates a local Matrix object:

```
#include "matrix.h" // IDL generated header file

#include "matrix_mgr.h" // local class implementation
```

```
.
.
.
Matrix *mlocal;
// Create a local Matrix object in this program
mlocal = new MatrixMgr(4, 3, 2, 1);
cout << "mlocal created:" << endl;
print(mlocal);
    .
    .
    .
```

#include ″matrix_mgr.h″
: To implement client-local objects, the application includes a local manager class that is derived from the interface class. Local code is also linked to the application that implements the client-local objects.

mlocal = new MatrixMgr(...)
: Clients create a local object by using the C++ **new** operator on the local manager class defined in the **matrix_mgr.h** header file.

Location Transparency of Local and Remote Objects

The previous sections showed separate cases of how to create objects that are either remote or local. However, many applications use a mixture of remote and local objects. For example, a presentation application can link in a video clip from another system (remote), or it can embed a copy of the video clip into the presentation itself (local). After the objects are created, we want the distinction to be as transparent as possible to simplify application code. In DCE, you can also intermix local and remote objects in function calls without needing to keep track of which is which.

To prepare your application to handle both local and remote objects simultaneously, do the following development steps:

1. Use the **cxx_static** ACF attribute to rename local versions of static functions.
2. Use the IDL compiler to produce both the client and server stub code.
3. Link into your client the client stub, server stub, and local object implementation code.

To accomplish this, you develop the client as if you are producing both a client and a server simultaneously. The only real difference is that you do not need any server initialization code. This means that your application includes the **idl**-generated manager class header and server stub, and manager implementation code for each interface. (See the following figure.) A client uses the client stub to produce and use remote objects. The client uses the server code to produce and use client-local objects of the interface class. This makes more sense when you think about server development: the manager class and code implement distributed objects of the interface class that are *local* to the server, so it helps to think local implementation code rather than server implementation code when we use server stubs, manager classes, and manager code in a client. If your application fails to use the server stub, the exception **rpc_x_no_server_stub** is raised by the client if your application tries to use local objects.

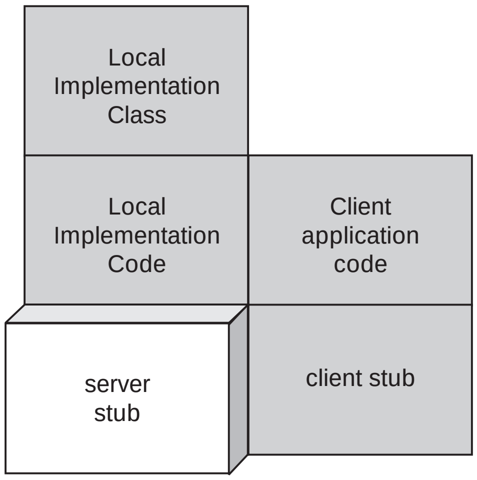


Figure 48. Clients Use the Server Stub

The IDL compiler requires an ACF such as the following when a client uses both remote and local objects:

```
/* FILE NAME: matrix.acf */

interface Matrix
{
    /* include files generated into the server stub */
    [sstub] include "matrix_mgr", "staticfunc";

    /* createMatrix must be mapped as a creator member function. */
    /* The argument MatrixMgr names the class that implements the */
    /* interface for the server stub.                */
    [cxx_new(MatrixMgr)] createMatrix();

    /* The "newMatrix" name represents the remote version of the */
    /* function that is used by either the client application or */
    /* the server stub.                       */
    [cxx_static(LocalMatrix)] newMatrix();
}
```

[sstub] include
The **include** statement causes the IDL compiler to include header files in stubs. Data structures and definitions in code that are required by stubs need to be included in this way. This example applies the **sstub** attribute to specify the inclusion of **matrix_mgr.h** and **staticfunc.h** files in the server stub only. The **matrix_mgr.h** file contains the definition of the client-local manager class. This class defines the implementation of the interface and is derived from the interface class. The file **staticfunc.h** contains declarations of static member functions for the interface. In this example, there is only one static member function: **LocalMatrix()**.

[cxx_new(MatrixMgr)] createMatrix();
The **cxx_new** attribute specifies that a static member function of the interface is an object creator function. An argument (in this case, **MatrixMgr**) is needed to name the manager class, the class derived from the interface class to implement local interface objects. For the client stub, the argument is ignored and the function **createMatrix()** is generated as a static member function. The client application uses this function to create a remote interface object. For the server stub (or, in this case, the client-local implementation), the **MatrixMgr** argument represents the manager class

name defined in a header file previously specified in the ACF with the **include** statement. The application uses the **new** operator on the **MatrixMgr** class to create a local interface object.

[cxx_static(LocalMatrix)] newMatrix();
The **cxx_static** attribute specifies the interface's static member functions. All static member functions need to have this attribute (unless you use the **static** keyword in the interface definition to specify the function as static). An argument is required to avoid name conflicts between the local and remote versions of the function when both client and server stubs are linked together in the same application. For the client stub, the argument is ignored and the client application calls **newMatrix()** for remote access to the interface. For the client-local (server) stub, the argument is used to name the function, and the application calls **LocalMatrix()** for local access to the interface.

The following example shows client code to create and use both remote and local objects from an interface class:

```
#include "matrix_mgr.h"
#include "printmatrix.h" // print() macro

void
main()
{
  idl_long_int d1, d2, d3, d4;
  Matrix     *mremote, *mlocal, *mr, *ml;

  d1 = 1; d2 = 2; d3 = 3; d4 = 4;

  cout << "Creating dynamic objects:" << endl;

  // Create a remote Matrix object on a server
  mremote = Matrix::createMatrix(d1, d2, d3, d4);
  cout << "mremote created:" << endl;
  print(mremote);

  // Create a local Matrix object in this program
  mlocal = new MatrixMgr(d4, d3, d2, d1);
  cout << "mlocal created:" << endl;
  print(mlocal);

  // Create another object from a local and remote one.
  // Whether the new matrix is local or remote depends on whether
  // the invoking object is local or remote.

  // create another remote Matrix while accessing a local object
  mremote->add(mlocal, &mr);
  cout << "mr is remote. It's the sum of mremote and mlocal:"
<< endl;
  print(mr);

  // create another local Matrix while accessing a remote object
  mlocal->add(mremote, &ml);
  cout << "ml is local. It is the sum of mlocal and mremote:"
<< endl;
  print(mr);

  // Applications should ALWAYS delete remote dynamic objects when
  // through, otherwise, the server will waste resources maintaining
  // them.
  delete mremote, mlocal, mr, ml;
  cout << "Client exiting" << endl;

  return;
}
```

Matrix *mremote, *mlocal, *mr, *ml;

Local and remote object references are both defined as pointers to the interface class. Depending on how an object is created, polymorphism causes the invocation of a client stub function for remote objects or the locally defined function for local objects.

mremote = Matrix::createMatrix(d1, d2, d3, d4);

mlocal = new MatrixMgr(d4, d3, d2, d1);
Clients call a static creator function to create a remote object on a server and use the C++ **new** operator to create a local object.

mremote->add(mlocal, &mr);
A client can use remote and local objects together. In this example, a local object (**mlocal**) is added to the invoking remote object (**mremote**) to create a new remote object (**mr**) that is the sum of the two.

mlocal->add(mremote, &ml);
In this example, a remote object (**mremote**) is added to the invoking local object (**mlocal**) to create a new local object (**ml**). Whether the resulting object is local or remote depends on the invoking object.

delete mremote, mlocal, mr, ml;
Clients use the C++ **delete** operator to delete both local and remote objects. If a client does not delete local objects prior to exiting, no real harm is done since all the memory for the application is released. However, clients should always delete remote objects when finished with them because the servers maintain them even after the client has exited.

Finding Known Remote Objects

Servers can register objects with the name service, such as the Cell Directory Service (CDS). Such objects are termed named objects. When an interface is compiled, the IDL compiler generates an overloaded **bind()** operation that allows a client to bind to a named object in several ways. These include the following:

- Bind by an object’s name
- Bind by an object’s UUID
- Bind by a binding handle

An *overloaded* operation’s argument list and functionality varies depending on which argument is used. The **bind()** operation of an interface is a static operation that returns a typed interface pointer. A zero is returned upon failure to locate and bind to the object.

Binding to Named Objects By Name

To bind to a named object by its CDS name, the argument provided to the **bind()** operation should be an **unsigned_char_t** pointer that specifies the name of the registered object in the CDS hierarchy. For example, the following code fragment uses a CDS name to create a local object proxy in the client application bound to a remote object:

```
Matrix m;

cout << "Binding to objects by name stored in CDS:" << endl;
m = Matrix::bind((unsigned_char_t *) "/.:/objects/identityMatrix");
if (m) {
    print (m);
} else {
    cerr << Cant bind to named object << endl;
}
```

In order for this to work, a server must have registered the object in CDS by calling the **register_named_object()** function.

Binding to Named Objects by UUID

To bind to a named object by its object ID, the argument provided to the **bind()** operation should be a **uuid_t** reference. The argument specifies the UUID of the registered object in the CDS hierarchy. The DCE environment variable **RPC_DEFAULT_ENTRY** must be set to indicate where the search for the object is to begin in the CDS name space. For example, the following code fragment uses an object's UUID to create a local object proxy in the client application bound to the remote object:

```
const char *UUID = "f063cf5a-c5c8-11ce-8a4b-08002be415b2";
Matrix    *m; // interface pointer
uuid_t    u; // uuid of named object
unsigned32  status; // error status

// get a uuid from string format
uuid_from_string ((unsigned_char_t *) UUID, &u, &status);
if (status !=uuid_s_ok) {
    // handle error case
}
// bind to a named object by uuid
m = Matrix::bind(u);
if (m) {
    print (m);
} else {
    cerr << Cannot bind to named object << endl;
}
```

Binding Explicitly to Known Objects

To bind to an object explicitly by its binding handle, the argument provided to the **bind()** operation should be a server binding handle of type **rpc_binding_handle_t**. Note that this method does not use CDS at all. For example, the following code fragment uses a binding handle to create a local object proxy in the client application bound to a remote object:

```
const char *UUID = "f063cf5a-c5c8-11ce-8a4b-08002be415b2";
const char *PROT = "ncacn_ip_tcp";
const char *HOST = "16.01.02.03";
const char *ENDP = "4041";

Matrix *m;  // interface pointer
unsigned_char_t *string_binding; // string binding
rpc_binding_handle_t binding_handle; // binding handle
unsigned32 status; // error status

// build a string binding from the various components
rpc_string_binding_compose(
     (unsigned_char_t *) UUID, // object uuid
     (unsigned_char_t *) PROT, // protocol sequence
     (unsigned_char_t *) HOST, // host address
     (unsigned_char_t *) ENDP, // transport endpoint
     NULL, // network options
     &string_binding,
     &status
);
if (status != rpc_s_ok) {
// handle error case
}

// convert a string binding into a binding handle
rpc_binding_from_string_binding(
    string_binding,
    &binding_handle,
    &status
);
if (status != rpc_s_ok) {
    // handle error case
}

m = Matrix::bind(binding_handle);
if (m) {
    print(m);
} else {
    cerr << "Cannot bind to named object" << endl;
}
```

rpc_string_binding_compose()
This RPC API routine combines string components of binding information into a single string representation of a binding.

rpc_binding_from_string_binding()
This RPC API routine creates a binding handle from a string representation of a binding handle.

m= Matrix::bind(binding_handle);
The **bind()** operation when used with a binding handle parameter binds to the object specified by the object's UUID and specific server binding information.

Multiple Interfaces and Interface Inheritance

Objects in useful applications are organized into groups (using classes) and hierarchies in order for people to more easily develop and maintain them. For the same reason, you use more than one IDL interface to logically group the objects and functionality of your applications. In addition, you can organize your interfaces into hierarchies that take advantage of the inheritance capabilities of C++ classes.

This discussion uses a traditional savings account example, as shown in the class hierarchy diagram of the following figure. First there is a high-level *Account* interface and then a *Savings* interface derived from the *Account* interface. The *Account* interface is specified separately from the *Savings* interface for the basic operations all accounts might have and to show how interface inheritance works. With this scheme, we can easily specify other kinds of accounts by using additional interfaces. (For example, we could also have a *Checking* interface.) Our example also has a separate *Loan* interface to show how to combine interfaces in applications. In the implementation of these interfaces, we derive a simple savings account class (**simpleSave**) from the savings interface, and we derive an overdraft-protected savings account class (**overdraft**) from both the savings and the loan interfaces.

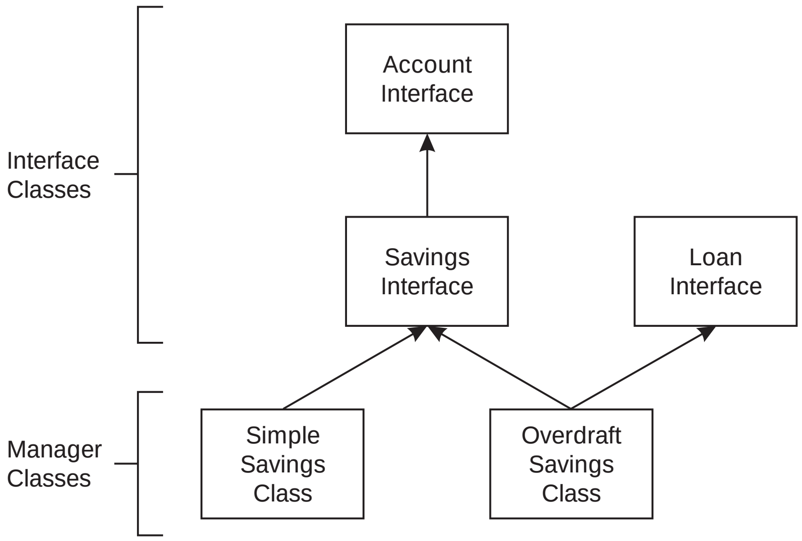


Figure 49. Multiple Interfaces and Inheritance

The Account interface contains the most basic operations for accounts, including one to obtain the account's balance, one to make deposits, and one to make withdrawals. This interface definition is as follows:

```
[
uuid(b3896a1c-8ee2-11ce-badc-08002b2bf322)
] interface Account
{
  double getAccountBalance();

  double deposit( /* Value returned is the balance. */
        [in] double amt
  );

  double withdraw(/* Value returned is actual amount withdrawn */
        [in] double amt
  );
}
```

Use the inheritance operator, **:**, in an interface definition to specify *interface inheritance.* In the following example, the Savings interface *inherits* operations from the Account interface. (Depending on your perspective, you can also say the Savings interface *is derived from* the Account interface.) When an interface inherits another, it also uses the **import** statement to be sure the operations and any data types of the inherited interface are available to the derived interface. The Savings interface definition is as follows:

```
[
  uuid(b388ab7c-8ee2-11ce-badc-08002b2bf322)
]  interface Savings : Account
{
    import "account.idl";

    static Savings * openSimple(
        [in] double  amt
    );
    static Savings * openOverdraft(
        [in] double  amt
    );

    double getSavingsBalance();

    void  setInterestRate(
        [in] double rate
    );

    void  addInterest();
}
```

The **openSimple()** and **openOverdraft()** static operations are object creator operations used to create new accounts on a server. Notice that the Account interface has no creator operations specified. This means that clients cannot create an Account object directly, but servers of course can. The non-static operations for the Savings interface include one to get the savings account balance (**getSavingsBalance()**), one to set the interest rate (**setInterestRate()**), and one to add the interest to the balance of the account (**addInterest()**)

In this application, we have decided that our server implements the overdraft account with a Loan interface. (Note we could have chosen to implement it in another way, and without an additional interface.) The Loan interface is not derived from another interface and is shown in the following example:

```
[
  uuid(912ef43d-8ee2-11ce-a54e-08002b2bf322)
]
interface Loan
{
  static Loan * openLoan(
    [in] double  amt,
    [in] double  rate,
    [in] long   months,
    [out] double  &payment
  );

  double getLoanBalance();

  void payment(
    [in] double amt
  );

  double recalculateLoan( /* returns payment amount required */
    [in] double  rate,
    [in] long   months
  );
}
```

The **openLoan()** operation is a static object creator operation to create a loan object. The **getLoanBalance()** operation gets the current balance of the loan and the **payment()** operation is used to make a payment on the loan. The **recalculateLoan()** operation sets new terms for the loan and returns the new monthly payment required.

There are no special techniques to follow in server initialization code except be sure that whatever is required for an individual interface is done for each interface your application uses. For example, the initialization code must be sure to register the endpoints for all interfaces.

Implementing Multiple Managers

Our implementation derives a simple savings account manager class (**simpleSave**) from the Savings interface class. Since the Savings interface is derived from the Account interface, all nonstatic operations in both interfaces must be declared in the manager class and defined in the manager code. Of course, additional functions and data types (such as constructors and destructors) can also be declared to specifically implement the interface.

Our implementation also derives an **overdraft** manager class for an overdraft type of savings account. The overdraft account has characteristics of both a savings account and a loan and demonstrates *multiple interface inheritance*. It is defined to have multiple inheritance by being derived from both the Savings and Loan interface classes.

Note: Applications can create C++ classes that inherit from multiple interface classes, but interface classes cannot inherit from multiple interfaces.

The following code shows the overdraft manager class and its implementation. This example has the manager implementation included within the class definition header file itself, rather than in separate C++ code. C++ allows you to combine implementation as part of the C++ class declarations. This is common practice when the implementation code for each member function is small.

```
#ifndef overdraft_i_h
#define overdraft_i_h
#include <iostream.h>
#include "savings.h"
#include "loan.h"

class overdraft : public Savings, public Loan {
  public:
    overdraft(idl_long_float amt)
    {
      balance = amt;
    }
    ~overdraft(void)
    {
        return;
    }
    idl_long_float getBalance()
    {
      return balance;
    }
//////////////// Member Functions from all interfaces ////////////////
    idl_long_float deposit(idl_long_float amt)
    {
        balance += amt;
        return balance;
    }
    void payment(idl_long_float amt)
    {
        balance += amt;
    }
    idl_long_float withdraw(idl_long_float amt)
    {
        balance -= amt;
        return amt;
    }
    void setInterestRate(idl_long_float r)
    {
        rate = r/loanTerm;
    }
    void addInterest()
    {
        balance += (balance * rate);
    }
    idl_long_float recalculateLoan(idl_long_float r, idl_long_int m)
    {
        if(balance < 0)
          {
          loanRate = r;
          loanTerm = m;
          return abs(balance) / loanTerm;
          }
        else
          return 0;
    }
    idl_long_float getAccountBalance()
    {
        return getBalance();
    }
    idl_long_float getSavingsBalance()
    {
        return getBalance();
    }
    idl_long_float getLoanBalance()
    {
        static idl_long_float loanBalance;
        if(balance < 0)
         loanBalance = abs(balance);
        else
         loanBalance = 0;
        return loanBalance;
    }
  private:
    idl_long_float balance = 0; //loan is automatic if negative balance
    idl_long_float rate   = 0.02; //2%
    idl_long_float loanRate = 0.15; //15%
    idl_long_int  loanTerm = 12;  //12 months
};
```

The manager class must declare all the nonstatic functions of all its inherited interfaces. These include all nonstatic operations defined in all three interfaces, including the Account, Savings, and Loan interfaces. Be sure to define the operation signatures exactly as they are declared in each **idl**-generated header file, or else the C++ compiler may not interpret the function as an implementation but rather as a new function. If this occurs, the class is interpreted as an abstract class, which means that your application cannot create instances of the manager class.

For this example, refer to the **savings_mgr.h** and **loan_mgr.h** header files generated by the IDL compiler to find the signatures of all the functions required. For example, the **deposit()**, **withdraw()**, and **getAccountBalance()** functions are from the Account interface but are redeclared in the derived Savings interface. The **payment()**, **recalculateLoan()**, and **getLoanBalance()** functions are declared in the Loan interface. The **setInterestRate()**,**addInterest()**, and **getSavingsBalance()** are declared in the Savings interface.

The static function implementations for the Savings and the Loan interface classes are not shown here but include **openSimple()**, **openOverdraft()**, and **openLoan()**.

Using Objects that Support Multiple Interfaces

When clients use objects whose interfaces are independent from each other, no special coding is required beyond the conventions described earlier: you just create, use, and delete objects for each interface. The most interesting circumstances involving multiple interfaces are those in which an object itself supports more than one interface.

Binding by Object Reference to Use a Different Interface

We defined our overdraft savings account to be derived from two different interfaces (see the following figure). However, the client does not have any knowledge of how a server implements the overdraft account. The client does have a way to create an overdraft account by calling the static function **openOverdraft()**, but that is defined in the Savings interface which has no access to the Loan interface. So how does an overdraft object inquire about its loan balance by using the Loan interface's **getLoanBalance()**member function, when the object reference is to the Savings interface? We obviously cannot simply create another object reference to the Loan interface and expect the two different object references to both refer to the same overdraft object.

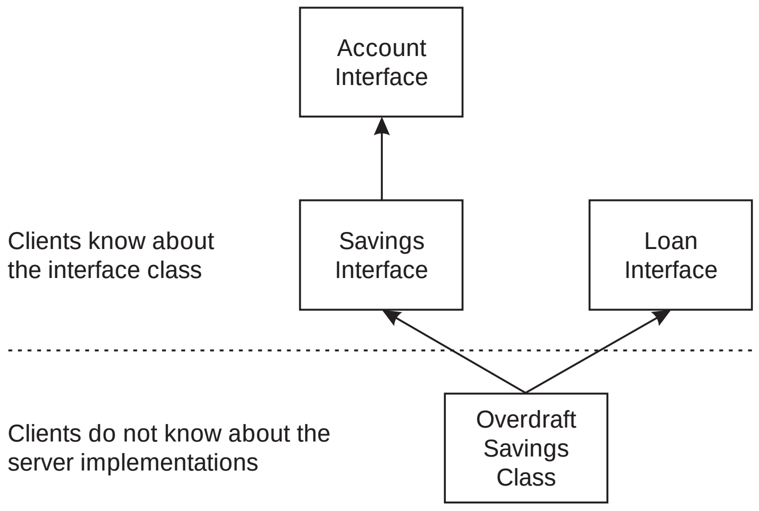

Account Interface
Clients know about the interface class
Savings Interface
Loan Interface
Clients do not know about the server implementations
Overdraft Savings Class

Figure 50. Clients Do Not Know About Server Implementations

The solution is to use an **idl**-generated member function. When the IDL compiler generates the interface classes, it also generates an additional **bind()** member function that allows the client to easily use other interfaces. The following examples show sample client code that creates and uses a new simple savings account object and an overdraft account object:

```
#include "savings.h"
#include "loan.h"

Account *a = 0;
Savings *ss = 0;
Savings *od = 0;
Loan  *iLoan = 0;
```

The interface classes are declared in the header files generated by the IDL compiler as follows:

```
ss = Savings::openSimple(456.12);
od = Savings::openOverdraft(568.19);
```

In this example, the client creates a new simple savings account object on a server by calling the **openSimple()** function. The function creates an object reference to the Savings interface. The client also creates a new overdraft account object on a server by calling the **openOverdraft()** function. This function also creates an object reference to the Savings interface.

A robust server would likely give clients a way to find accounts again later by making the objects named and persistent; but, to simplify our examples, we use only dynamic objects. Therefore, accounts must be recreated each time a client runs.

```
balance = ss::getSavingsBalance();
assert(balance == 456.12);
balance = od::getSavingsBalance();
assert(balance == 568.19);
```

Object references to the Savings interface can call any member functions of the Savings and Account interfaces as follows:

```
iLoan = Loan::bind(od);
```

To use a different interface, clients use the built-in **bind()** member function with an object reference parameter. In this example, the function creates an object reference to the Loan interface, **iLoan**, from the object reference to the Savings interface, **od**:

```
balance = iLoan->getLoanBalance();
cout << "Loan Balance: " << balance << endl;
```

The object can now call any member function of the Loan interface as follows:

```
ss->deposit(20.01);
```

An object reference to one interface can access member functions of its inherited interfaces, as expected. In this example, **ss** is an object reference to a Savings object, but the **deposit()** function is specified in the Account interface:

```
balance = ss->getAccountBalance();
cout << Balance: " << balance << endl;
a = Account::bind(ss);
balance = a->getAccountBalance();
cout << Balance: " << balance << endl;
```

As an aid to debugging, it is a good idea to use the interface in which the operation is declared, even if the inherited operation can be resolved. When the object calls **getAccountBalance()** with a Savings object reference, the function is executed in the client stub for the Savings interface. On the other hand, when the same function is called with an Account object reference, the Account client stub function is executed.

Finding Out if an Interface is Supported

One of the most common reasons to find out if an interface is supported is to determine whether or not a new version of an application uses an additional interface. The new clients must check for application compatibility by inquiring as to whether the new interface is supported. Compatibility is easily tested by calling the **idl**-generated **bind()** function with an object reference parameter, as described in the previous section. An interface is not supported if the returned result is 0 (zero). This simple test implies that it is easy to create new versions of applications by adding additional interfaces, rather than running the risk of creating incompatibility by modifying existing interfaces.

The following example shows how to inquire if an interface is supported. Suppose we are told that some servers on our network implemented the overdraft account without using the Loan interface. This would not prevent our clients from creating and using overdraft objects; we would just not have the Loan interface to use to inquire about the status of an overdraft. In this scenario, the client that has the Savings interface could inquire as to whether the object also supports the Loan interface, as in the following:

```
iLoan = Loan::bind(ss);
if(iLoan == 0)
  cout <<"Simple accounts do not support the Loan interface."<< endl;
iLoan = Loan::bind(od);
if(iLoan == 0)
  cout <<
   "This overdraft account doesn't support the Loan interface." << endl;
```

In the first case, attempting to bind a simple savings object to the Loan interface should always return 0 (zero). In the second case, if a zero value is returned, this overdraft object does not support the Loan interface (the Loan interface is not inherited).

Multiple Interfaces and Local Objects: a C++ Enhancement

The **bind()** member function that takes an object reference also works for local interfaces and objects. This means that you can use IDL to specify, implement, and test combinations of local interfaces without the overhead of remote procedure calls. You may find this a useful approach when designing, prototyping, and debugging your interfaces and implementations. The steps are as follows:

1. Create your IDL files by using the **uuid** and **local** attributes in the interface headers.
2. Use the IDL compiler with the **-lang cxx** option to compile the interfaces. The IDL compiler generates only the header files for each interface when the interface has the **local** attribute. No stubs are generated.
3. Develop the manager class, manager implementations, and static member functions as you would for typical servers.
4. Create a client that includes the **idl**-generated header files. The client also calls the **idl**-generated **bind()** function that binds by object reference, to switch between local interfaces.
5. Link together the machine object code for the client, manager class, manager code, and static functions.
6. Test the client application program.

Passing C++ Objects as DCE RPC Parameters

IDL allows the passing of any C language basic or constructed data type as an RPC parameter, mainly through the use of attributes. However, the C++ language makes it much easier and convenient for the programmer to define new types using class definitions. A C++ application can contain a wealth of class definitions modeled after real world objects, usually in the form of class libraries. The implementation details of a class library definition are hidden from the programmer in favor of a public interface or set of operations to manipulate the class instance. In addition, software vendors are in the business of providing class libraries containing all sorts of class definitions that are ready to use by the application programmer.

As applications move towards the client/server model, and as distributed object technology becomes the vehicle for such a model, RPC must be able to pass C++ objects as parameters efficiently and intrinsically.

When an application is distributed, a number of issues arise that must be dealt with. These issues include the ability of the network to pass large amounts of data, the problem of passing pointers as RPC arguments, and the differences in the representation of a piece of data in the computer's memory that results from different machine architectures. These problems are addressed by DCE implementations adhering to the network data representation (NDR) for data types and the effective use of attributes in the interface definition. However, other problems that are specific to the C++ language include the following:

Data Hiding

One advantage of a class definition is that it allows the application designer to model a programming language construct after some real world object and to interact with the construct in a high level fashion. The details of how the construct is built and manipulated should be handled by the designer of the class. The application programmer should be insulated from the class internals and only needs to be aware of the public interface to the object. However, this programming model exposes a fundamental problem when extended to DCE RPC. In passing a parameter to a remote procedure, the DCE runtime library must be able to marshal the RPC parameters over the network on behalf of the caller of the remote operation and unmarshal those parameters and reconstruct the data type on the server side of the application. If users are able to create new and exotic data types, how can the DCE runtime know how to marshal them? It is unreasonable to expect that DCE could be extensible enough to track and know how to marshal new data types as they are created. It is also unreasonable to expect class designers to supply their own support for marshaling objects. This is especially true for data types that are provided as class libraries by outside vendors having no connection at all with DCE.

Inheritance

Inheritance and polymorphism are techniques available with C++ whereby a generic class type is used by an application but the actual object is created from a more specific class type. A classic example of this is a generic class called Shape and a number of specific classes such as Circle, Square and Cylinder that all derive from the Shape class. Shape might have operations such as **draw()** and **rotate()** which cause the object to be drawn onto the screen and rotated. The application can have an array of Shape objects and cause each one in turn to be displayed and manipulated. However, the array could be a mix of Circle, Square and Cylinder objects. Each object will know how to draw and rotate itself. For some objects, a function such as **rotate()** may have no meaning. The polymorphic behavior of the Shape class will forward the **draw()** operation to the correct specific drawing operation implemented by the object. The **rotate()** operation behaves similarly for classes that support rotation. And if the object does not support rotation, the Shape class will supply its own rotate implementation which may actually do nothing.

The problem of passing a Shape object is that the DCE runtime may not know what type of shape the object really is. A Shape could have some self-identifying information, but this will often not be the case. Furthermore, if there were some shape identifier, it would need to track new class types as they are introduced into the application. This type of design is not very extensible and contrary to the object oriented methodology.

Lots of data

Another problem with passing a C++ object over the network is fundamental to any RPC argument. As the amount of data needed to be passed over the wire and recreated in the server process increases, the performance of the RPC call will obviously decrease. The decision as to what kinds of operations are remote and what types of data they require is a basic design issue. For example, consider a stack type. If the stack is small then it may be advantageous to pass the entire stack over the wire, recreate it on the server side, allow the server to update it, and then pass it back to the client side so that it reflects any updates the server made to it. The IDL language supports such a paradigm by using an array network type along with other parameters to indicate the array size. However, this paradigm quickly breaks down as the stack size increases. A better way for the server to access a large stack would be to pass a stack reference to the server and allow the server to access the stack by making RPC calls to it.

Two programming methodologies are presented to illustrate how C++ objects can be passed as DCE RPC parameters: data representation and delegation. It is a design choice as to which solution better applies to a specific application problem. By using these methodologies, class libraries can be easily integrated into an application. Both solutions are intended to be handled primarily at the interface definition level so that the application itself can be designed in a normal and natural way while minimizing the issue of distributing the application.

Representation

The DCE IDL compiler supports a feature to allow a network representation of a data type to differ from the representation used by the application. This feature is invoked by using the **represent_as** attribute on a data type in the Attribute Configuration File (ACF). Applying this attribute to an IDL data type allows the network representation of a data type to be isolated within the generated stubs. The programmer is required to supply four conversion routines when using this feature. The function signatures for these four routines are generated by the IDL compiler. Their purpose is to convert an RPC argument from the application presented type to the network type, convert from the network type to the presented type, and to free memory used by the network and presented types. Presumably, a class library designer could supply the four conversion routines along with the IDL generated stub routines as a library. In this way, the application programmer need not be aware of how the data is transmitted across the wire nor that the conversions take place.

For example, consider the C++ String class which is commonly supplied by C++ compiler vendors or easily implemented by the programmer. The IDL compiler has no notion of a String class since it is not a primitive or constructed IDL type. The class definition must be made known to the IDL compiler by using the include directive to include the class definition into the generated header file. But the DCE runtime does not know how to marshal the String type since its internals are hidden and, in fact, could very well differ in its implementation between vendors.

To allow a String type to be passed as an RPC argument, a network type for a String object is defined in the IDL file to be an array of characters with the string attribute applied. An ACF file is then created for the interface to apply the **represent_as** attribute to the network type. The following code fragment is for the IDL file represent an IDL character array as a String class in an application:

```
[ uuid(c5a7c094-c5e3-11ce-bac2-08002be415b2) ]
IText {
    typedef [string,unique] char * net_string; /* 1 */
    static net_string toUpper([in] net_string s); /* 2 */
    static net_string toLower([in] net_string s); /* 3 */
}
```

The following code is an ACF definition for a String type:

```
{
    include "String";  /* 4 */

    typedef [represent_as(String)] net_string; /* 5 */
}
```

The code is described as follows:

1. A net_string is defined to be a unique pointer to a string
2. The static **toUpper()** operation takes a String argument and returns another one
3. The static **toLower()** operation takes a String argument and returns another one
4. Include the **String.h** file into the **idl**-generated header file
5. The network type net_string is presented as the C++ String type in the application

Using automatic binding, the client application would invoke the static **toUpper()** operation as follows:

```
String s1("Hello, World"); // create a local String object
String s2 = IText::toUpper(s1); // RPC call returns another
                // String object
```

Note that a unique attribute is specified for the net_string type. Unique pointers should always be used when the **represent_as** attribute is applied to a pointer type.

The routines to convert between the network type and the presented type are automatically invoked by the DCE runtime during the marshalling and unmarshalling process. The IDL compiler generates a function signature to free the presented data type. In this example, this routine would be named **net_string_free_local(String *)**. The purpose of this routine is to free the memory occupied by the stack variable in the server stub that represents the RPC parameter. But since the C++ compiler will generate code to delete local stack objects when the server stub routine is exited, this routine should not free its argument.

The **represent_as** attribute is properly used when the data comprising the C++ object can be represented by some primitive or constructed IDL data type. The object's data must be accessible by the application. The overhead involved with using the **represent_as** attribute is the conversion from one type to the other and the freeing of memory.

It may not always be advantageous to use the **represent_as** attribute to pass a C++ object as an RPC parameter. Consider the case presented earlier where a generic Shape class is used in a class hierarchy with the more specific Square, Circle, and Cylinder shapes derived from it. An application may wish to pass a Shape object as an RPC parameter. Using the **represent_as** feature would require the conversion routines to convert from a shape to some NDR structure that can be defined in IDL. However, this is complicated by the fact that one IDL type may not be sufficient to represent all possible shapes. To solve this, a discriminated union of different shape types could be defined. But it is also very possible that the internals of the classes are not exposed to the application. The user may have no knowledge of what data types are needed to represent even the simplest shapes such as a square. Furthermore, as new shapes are introduced into the application, the conversion routines would also require extensions to handle the new shapes. An object oriented application should be extensible without requiring such overhead.

Another drawback to using conversion routines is efficiency. Consider a common C++ Stack class and a distributed implementation of a reverse Polish notation algorithm. The algorithm maintains a stack of operands. When an operator is processed, the required number of operands are popped off the stack, the operation is performed on them, and the result is pushed back onto the stack. For this example, let's assume that the algorithm supports the **plus()**, **minus()**, **multiply()** and **divide()** binary operations. In order to illustrate the distributed nature of the algorithm, we can further assume that the client reads an equation in reverse Polish notation from standard input and maintains the stack locally, but the binary operations are implemented within a remote server process. Hence, the server process needs access to the same stack as the client. Simply passing the stack to the server process in its entirety would be inefficient since only the top two elements need to be accessed per operation. A large stack would quickly degrade the performance of the algorithm, especially since the stack would have to be passed as both an input and output parameter.

Delegation

An alternative to passing the stack is to treat the stack as a distributed object and pass a reference to it. The server and client would have access to the single stack in the application and the server could use the stack object reference to push and pop elements. A DCE distributed object requires that there be an interface defined for the object and the object implementation be derived from the generated interface class. If the stack being used is supplied by a third party vendor, it may not be possible to modify its definition to derive it from an IDL generated class. The solution is to create a delegate class for the stack to act as an interface to the actual stack object. A delegate class encapsulates the real object and forwards operation invocations to it. The IDL language has been extended to include the ACF attribute **cxx_delegate** to take advantage of this idiom.

An interface using this attribute will cause the generated interface class to wrap the real object. Only the operations that need to be remote need to be defined in the delegate class interface definition file. The application would then link the delegate server stub with the client. Likewise, the delegate client stub would be linked with the server. The DCE runtime will transparently perform the necessary setup to allow the client application to act as a server for the delegate class. The following example illustrates the use of the **cxx_delegate** attribute with a Stack class definition:

```
[ uuid(0ea74f20-e2dc-11ce-9a8e-08002be415b2) ]
interface IStack  /* 1 */
{
    void push([in] double x); /* 2 */
    double pop(); /* 3 */
}
```

The ACF definition for delegation of the IStack interface is as follows:

```
[ cxx_delegate(Stack) ] /* 4 */
interface IStack{
    include "Stack"; /* 5 */
}
```

1. An IStack interface is defined
2. The **push()** operation pushes an element onto the encapsulated stack
3. The **pop()** operation pops an element from the encapsulated stack
4. The IStack interface is a delegate for a Stack class
5. include the **Stack.h** file into the IDL generated header file

Using the generated server stub and header file from the above IDL fragment, the client application would instantiate an IStack interface pointer and pass it to the remote procedure as follows:

```
Stack s; // create a local stack object
IStack *iStack;  // declare an interface pointer to the local stack
iStack = new IStackMgr(&s); // create the interface ptr
         // using the local
stack
```

The **cxx_delegate** attribute causes the IDL generated classes to be built slightly different than a normal interface class. The interface class contains a constructor that takes a pointer to the delegated class instance as an argument and the manager class supplies complete function bodies. The programmer does not need to supply a manager class for an interface using this attribute.

The server application would use the interface pointer to invoke the **push()** and **pop()** operations on the client's stack instance. The overhead involved is the remoteness of the **push()** and **pop()** operations which are implemented as RPC calls from the server to the client. In this example, the client application would be linked with the IDL generated server stub from the IStack interface and the server application would be linked with the IDL generated client stub from the IStack interface. No extra DCE API calls are required on the part of the client or server stubs. The DCE runtime will handle the necessary overhead to allow the client application to act as a server for the IStack interface.

This idiom is most effectively used when a class type is needed as an RPC argument but the class hierarchy can not be changed by the application or when the overhead of the RPC calls to access the object is outweighed by the combined overhead of converting the object to a network type and the complexity of passing or updating a large amount of data in the RPC call.

Integrating C and C++ Clients and Servers

This chapter has assumed your clients and servers are both written in C++, and the rest of this guide describes how to write clients and servers that are both written in C. Two fundamental differences between these types of applications are their perception of what interfaces represent and whether clients bind to servers or objects.

For C applications, the model tends to be functionally oriented. The important features are the operations, in which an interface represents a convenient set of operations with associated data structures. Clients bind to servers that support the set of operations and data.

For C++ applications, the model tends to be object oriented. In this model, the important feature is the interface itself, which represent a class of objects. Operations are an integral part of each object, but data structures tend to be hidden in the implementations on the servers and not exposed in the interface. In this model, clients bind to objects that support the interfaces.

This section addresses the intersection of these two models in the following ways:

- Writing C++ clients with a functional approach so that they bind to servers (written in C) rather than to distributed objects
- Writing C clients so that they can bind to distributed objects rather than to servers

Writing a C++ Client for C Servers

Suppose you are writing a C++ client that needs to use an interface definition that has not taken advantage of the IDL C++ features. A logical example is an older interface definition written prior to the introduction of the C++ features of OSF DCE Version 1.2. An older interface is not designed to specify a class of objects and the associated member functions. This means that servers for older interfaces do not maintain objects in the way described in this chapter (if they maintain objects at all).

This section uses the following simplistic interface for demonstration:

```
[
  uuid(166ab38b-95f9-11ce-9387-08002b2bf322)
]  interface old_interface
{
    double op1();

    void  op2([in] long input);

    void  op3();
}
```

If you simply compile this interface definition for the C++ language and build the C++ client application, the application cannot invoke any of the old interface's member functions because no object can exist on a server. However, static member functions do not require an object in order to invoke them, so the solution is to make the operations of an older interface static member functions.

In order for your C++ client to use an older interface, perform the following steps:

1. Create an ACF for the interface and apply the **cxx_static** attribute to *every* operation of the interface. For example:

```
interface old_interface
{
    [cxx_static] op1();
    [cxx_static] op2();
    [cxx_static] op3();
}
```

2. Use the IDL compiler with the **-lang cxx** option to compile the interface and generate the header files and C++ stubs. Link the code into your C++ client application as usual.
3. Call the static functions where needed in the C++ client application by using the scope operator (::). For example:

```
#include "old_interface.h"

main()
{
idl_long_float result;
idl_long_int input = 1;

result = old_interface::op1();
old_interface::op2(input);
old_interface::op3();
return 0;
}
```

Writing a C Client for C++ Servers

If you wish, you can develop C language clients that use interfaces written with C++ features. Whenever the interface definition is compiled for C++, C structures, macros, and function prototypes are automatically built into the header file and stubs to give this capability.

For example, the following **get()** operation is defined in the Matrix interface definition:

```
long get(
long row,
long col
);
```

The macros generated by the IDL compiler are formed by combining the name of the interface and the name of the operation with an underscore between. For example, to allow a C client to invoke the **get** operation on the interface, the IDL compiler generates the following macro in the header file:

```
Matrix_get(obj, row, col)
```

Since member functions cannot be called in C with an implied object (the C++ *this* object), each member function for the C macros has an additional object argument as the first parameter. The remaining arguments are the same as those specified in the IDL input file.

To obtain the interface pointer using the C mapping, use one of the bind routines generated by the IDL compiler for the C interface. These are also generated in the header file. For example, the Matrix interface supports the following C macros for binding to a remote object:

```
Matrix *Matrix_bind_by_name(unsigned_char_t *name);
Matrix *Matrix_bind_by_uuid(uuid_t * u);
Matrix *Matrix_bind_by_hndl(rpc_binding_handle_t bh);
```

All static member functions of an interface are also supported for C. The macros are formed in a manner similar to the normal member functions (by joining the interface name and the operation name with an underscore), except there is no need for an additional argument to represent a current object. For example, if the Matrix interface supports the **createMatrix()** static operation, the following example C code invokes the operation:

```
/* code fragment showing the use of C macros */
Matrix *m; /* a C structure to represent an interface */
/*
** invoke a static member function to get an interface
** pointer and invoke operations on it.
*/
m = Matrix_createMatrix(1, 2, 3, 4);
if (!m) {
/* handle error */
} else {
printf("[%d, %d]\n", Matrix_get(m, 0, 0), Matrix_get(m, 0, 1));
printf("[%d, %d]\n", Matrix_get(m, 1, 0), Matrix_get(m, 1, 1));
}
```

Chapter 16. Writing Internationalized RPC Applications

An internationalized DCE RPC application is one that

- Uses the operating system platform's locale definition functions to establish language-specific and culture-specific conventions for the user and programming environment.
- Isolates all user-visible messages into message catalogs by using the **sams** (symbols and message strings) utility.
- Uses the DCE general-purpose application messaging routines, **dce_msg_ *()** and **dce_svc_ *()**, to display all program messages.
- Uses DCE RPC-provided or user-defined character and code set evaluation and automatic conversion features to ensure character and code set interoperability during the transfer of international characters in remote procedure calls between RPC clients and servers.

A locale defines the subset of a user's environment that depends upon language and cultural conventions. A locale consists of categories; each category controls specific aspects of some operating system components' behaviors. Categories exist for character classification and case conversion, collation order, date and time formats, numeric nonmonetary formatting, monetary formatting, and formats of informative and diagnostic messages and interactive responses.

The locale also determines the character sets and code sets used in the environment. The syntax and use of a locale definition function depends on the operating system platform in use with DCE. See your operating system programming guide and reference documentation for a description of the system's locale definition functions and locale categories.

The **sams** utility provides DCE services and application programs with a method for defining and cataloging user-visible messages, while the DCE messaging functions allow DCE services and application programs to display messages in a consistent manner. "Chapter 3. DCE Application Messaging" on page 37 describes how to develop an application that uses the DCE messaging routines and how to use the **sams** utility to create and generate message catalogs. See the *OSF DCE Application Development Reference* for a description of DCE messaging routine syntax, and the **sams(1dce)** reference page for a description of **sams** usage.

The remainder of this chapter describes the DCE RPC features for character and code set interoperability in remote procedure calls that are available to programmers who are developing internationalized DCE RPC applications. The first section describes the concepts of character sets, code sets and code set conversion and explains the default character and code set conversion mechanism that the RPC runtime protocol supports for remote procedure calls. The remaining sections describe the execution of a remote procedure call when it uses the DCE RPC features for character and code set interoperability, and explains how to build an RPC application that uses these features.

Character Sets, Code Sets, and Code Set Conversion

A character set is a group of characters, such as the English alphabet, Japanese Kanji, and the European character set. To enable world-wide connectivity, DCE guarantees that a minimum group of characters is supported in DCE. The DCE RPC communications protocol ensures this guarantee by requiring that all DCE RPC clients and servers support the DCE Portable Character Set (PCS). The *Introduction to OSF DCE* lists the characters in the DCE PCS. The IDL base type specifiers **char** and **idl_char** identify DCE PCS characters.

A code set is a mapping of the members of a character set to specific numeric code values. Examples of code sets include ASCII, JIS X0208 (Japanese Kanji), and ISO 8859-1 (Latin 1). The same character set can be encoded in different code sets; consequently, DCE can contain RPC clients and servers that use the same character set but represent that character set in different numeric encodings. Code set conversion is the ability for a DCE RPC client or server to convert character data between different code sets.

The DCE RPC communications protocol, through the NDR transfer syntax, provides automatic code set conversion for DCE PCS characters encoded in two code sets: ASCII and EBCDIC. The RPC communications protocol automatically converts character data declared as **char** or **idl_char** between ASCII and EBCDIC encodings, as necessary, for all DCE RPC clients and servers.

The DCE RPC communications protocol does not provide support for the recognition of characters outside of the DCE PCS, nor does it provide automatic conversion for characters encoded in code sets other than ASCII and EBCDIC.

However, DCE RPC does provide IDL constructs and RPC runtime routines that programmers can use to write RPC applications that exchange nonPCS, or international, character data that is encoded in code sets other than ASCII and EBCDIC. These features provide mechanisms for international character and code set evaluation and automatic code set conversion between RPC clients and servers. Using these features, programmers can design their applications to run in a DCE that supports multiple heterogeneous character sets and code sets.

The next section describes the remote procedure call execution model when the DCE RPC features for character and code set interoperability are used.

Remote Procedure Call with Character/Code Set Interoperability

Table 8 on page 150 in "Chapter 11. Developing a Simple RPC Application" on page 149 illustrates the basic tasks of an RPC application. Table 12 shows these basic tasks integrated with the additional tasks required to implement an RPC that provides character and code set interoperability.

Table 12. Tasks of an Internationalized RPC Application

Client Tasks	Server Tasks
	1. Set locale.
	2. Select network protocols.
	3. Register RPC interfaces.
	4. Advertise RPC interfaces and objects in the namespace.
	5. Get supported code sets and register them in the namespace.

Table 12. Tasks of an Internationalized RPC Application (continued)

Client Tasks	Server Tasks
	6. Listen for calls.
7. Set locale.	
8. Establish a character and code set evaluation routine.	
9. Find compatible servers that offer the procedures.	
10. **Call the remote procedure**.	
11. Establish a binding relationship with the server.	
12. Get code set tags from the binding handle.	
13. Calculate the buffer size for possible conversion of input arguments from a local to a network code set.	
14. Convert input arguments from a local to a network code set (if necessary).	
15. Marshall input arguments	
16. Transmit arguments to the server's runtime.	
	17. Receive a call.
	18. Get code set tags sent from the client.
	19. Calculate the buffer size for possible conversion of input arguments from network to local code set.
	20. Unmarshall input arguments.
	21. Convert input arguments from a network to a local code set (if necessary).
	22. Locate and invoke the called procedure.
	23. **Execute the remote procedure**
	24. Calculate the buffer size for possible conversion of output arguments from a local to network code set
	25. Convert output arguments from a local to a network code set (if necessary).
	26. Marshall output arguments and return value.
	27. Transmit results to the client's runtime.
	28. Remove code set information from namespace on exit.
29. Receive results.	
30. Calculate the buffer size for possible conversion of output arguments from a network to a local code set.	
31. Unmarshall output arguments.	
32. Convert output arguments from a network to a local code set (if necessary).	
33. Pass to the calling code the results and return control to it.	

As illustrated in the table, the internationalized RPC execution model consists of the following new steps:

1. Both client and server invoke a platform-dependent function to set their locale during initialization. This step establishes the client's and the server's local

character and code set; that is, the character and code set currently in use by processes on the client host and processes on the server host.

2. The server, as part of its initialization phase, calls a DCE RPC routine that retrieves information about code sets support on the server's host. The RPC routine examines the host's locale environment and its code set registry to determine the host's supported code sets; that is, code sets for which conversion routines exist that processes on the host can use to convert between code sets, if necessary.

   The code set registry is a per-host file that contains mappings between string names for the supported code sets and their unique identifiers. OSF assigns the unique identifiers for the code sets and DCE licensees, and DCE administrators assign their platform string names for the code sets. The DCE RPC routines for character set and code set interoperability depend upon a code set registry existing on each DCE host. For more information about the code set registry, see the *OSF DCE Administration Guide—Introduction* and the **csrc(8dce)** reference page.

   The routine returns a list of the supported code sets to the server; the list consists of each code set's unique identifier.

3. The server next calls a new RPC NSI routine to register the supported code sets information in the name service database. Recall that a server can use the NSI to store its binding information (information about its interfaces, objects, and addresses) into its own namespace entry, called a server entry. The new RPC NSI routine adds the supported code sets information as an attribute that is associated with the server entry, which the server created when it used the NSI export operation to export its binding information into the name service database. Placing the code sets information into the name service database gives RPC clients access to this information.

4. Before it calls the RPC NSI routines that locate a server that offers the desired remote procedure, the client calls a new RPC routine that sets up a character and code sets compatibility evaluation routine.

5. The client calls RPC NSI routines to locate a compatible server. The RPC NSI routines invoke the character and code set compatibility evaluation routine set up by the client to evaluate potential compatible servers for character and code set compatibility with the client.

6. The evaluation routine imports the server's supported code sets information from the name service database, retrieves the client's supported code sets information from the client host, and compares the two. If the client and the server are using the same local code set—the code set that processes on the host use to encode character data—then no code set conversion is necessary, and no data loss will result.

   If client and server are using different local code sets, then it is possible that the server is using a different character set than the client. The client does not want to bind to a server that is using a different character set, since significant data loss would result during character data conversion. Consequently, the evaluation routine uses the server's code set information to determine its supported character sets, and rejects servers using incompatible character sets. For example, if a client is using the Japanese Kanji character set (such as JIS0208), the evaluation routine rejects a server that offers the desired remote procedure but which is using the Korean character set (such as KS C 5601).

   If the client and server are character set compatible, and they support a common code set into which one or the other (or both) can convert, the evaluation routine deems the server to be compatible with the client. The NSI import routines return this server's binding information to the client.

7. The client makes the remote procedure call.
8. A client stub is called, with the character data represented in the local form and in the local code set.
9. Before marshalling the input arguments, the client stub calls a new stub support routine that retrieves code set identifying information that the evaluation routine established in the binding handle.
10. The client stub next calls a new stub support routine that determines, based on the code set identifying information, whether the character data needs to be converted to another code set and, if so, whether the buffer that currently holds the character data in the local form and code set is large enough to hold the data once it is converted. If the routine determines that conversion is necessary and a new buffer is required, it calculates the size of that buffer and returns the value to the client stub.
11. The client stub next calls a new stub support routine that converts, based on the code set identifying information, the character data from the local code set to the appropriate code set to be used to transmit the data over the network to the server (called the network code set).
12. The client stub then marshalls the input arguments and transmits them to the server runtime along with code set identifying information.
13. The server stub is called, with the character data represented in the network form (which is always **idl_byte**) and in the network code set.
14. The server stub unmarshalls the input arguments.
15. The server stub next calls a new stub support routine that determines, based on the code set identifying information passed in the client call, whether the character data needs to be converted from the network code set to the server’s local code set and, if so, whether the buffer that currently holds the character data in the network format and code set is large enough to hold the data once it is converted. If the routine determines that conversion is necessary and a new buffer is required, it calculates the size of that buffer and returns the value to the server stub.
16. The server stub next calls a new stub support routine that converts, based on the code set identifying information, the character data from the code set used on the network to the server’s local code set.
17. The server stub invokes the manager routine to execute the remote procedure.
18. Before marshalling the results of the remote procedure (the output arguments and return values), the server calls a new stub support routine to determine whether conversion from the server’s local code set is necessary, based on the code set identifying information it received from the client, and whether or not the buffer currently holding the character data is large enough to accommodate the converted data. If a new buffer is required, the stub support routine calculates the size of this new buffer and returns it to the server stub.
19. The server stub next calls a new stub support routine that converts, based on the code set identifying information from the client, the character data from the server’s local code set to the network code set.
20. The server stub marshalls the converted output arguments and transmits them to the client runtime along with code set identifying information.
21. The server initialization procedure also contains a call to a new RPC routine that removes the code set information from the server entry in the name service database if the server exits or is terminated.
22. The client stub is called, with the character data in network format and code set.
23. The client stub unmarshalls the output arguments.

24. The client stub next calls a new stub support routine that determines, based on the code set identifying information passed by the server, whether the character data needs to be converted from the network code set to the client's local code set and, if so, whether the buffer that currently holds the character data in the network format and code set is large enough to hold the data once it is converted. If the routine determines that conversion is necessary and a new buffer is required, it calculates the size of that buffer and returns the value to the client stub.
25. The client stub next calls a new stub support routine that converts, based on the code set identifying information, the character data from the code set used on the network to the client's local code set.
26. The client stub passes the data to the client in the local format and code set.

Note that the stub conversion routines do not implement code set conversion. Instead, they call POSIX compliant **iconv** code set conversion routines, which are part of the local operating system. As a result, if the platform to which DCE is ported does not provide these POSIX conversion routines, DCE applications that run on this platform cannot use the DCE RPC character and code set interoperability features.

Building an Application for Character and Code Set Interoperability

An application programmer who wishes to design his or her RPC application for character and code set interoperability performs the following steps:

1. Writes the interface definition file (**.idl**) to include constructs that will enable automatic code set conversion during remote procedure execution.
2. Writes an associated attribute configuration file (**.acf**) for the interface that includes ACF attributes that will enable automatic code set conversion during remote procedure execution.
3. Writes the stub support routines that client and server stubs use to carry out automatic code set conversion during a remote procedure call. You can omit this step if you use the stub support routines supplied with DCE.
4. Writes the server code and includes the steps to get the server's supported code sets and export them to the name service database, and to remove them from the name service database upon termination or exit.
5. Writes the client code and includes the steps to set up the character and code set evaluation mechanism.
6. Writes the character and code set compatibility evaluation routine. You can omit this step if you use one of the evaluation routines supplied with DCE.

Note that building an RPC application for character and code set interoperability imposes some restrictions on the application. For example, an application that uses the RPC character and code set interoperability features cannot use customized binding handles. See “Chapter 18. Interface Definition Language” on page 357 for more details on internationalized RPC application restrictions.

The next sections describe the steps just outlined in more detail.

Writing the Interface Definition File

The interface definition file is where the set of remote operations that constitute the interface are defined. The first step in writing an interface definition file that supports automatic code set conversion is to create a special **typedef** that, when used in operation parameters, represents international character data that can be automatically converted, if necessary, before marshalling and unmarshalling at client and server sites.

As described in “Chapter 17. Topics in RPC Application Development” on page 313, the data representation for a **byte** data type is guaranteed not to change when the data is transmitted by the RPC communications protocol. Consequently, the special international character data type defined in the **.idl** is always declared to be a **byte** type so that the RPC protocol will not automatically treat it as a DCE PCS character and convert it between ASCII and EBCDIC.

The second step in writing an interface definition file that supports automatic code set conversion is to define, for each operation that will transmit the special international character data type, a maximum of three operation parameters that will tag the international characters being passed in the operation’s input and output parameters with code set identifying information established during the client-server evaluation and binding procedure. These parameters are the following:

- The sending tag, which indicates the code set the client is using for international characters it transmits over the network. The sending tag has the **in** parameter attribute and is applied to international character data declared in the operation’s input parameters. If the operation does not specify any international character data as input, then it is not necessary to create this parameter.
- The desired receiving tag, which indicates the code set in which the client prefers to receive international character data sent back from the server as output. The desired receiving tag has the **in** parameter attribute. If the operation does not specify any international character output data, then it is not necessary to create this parameter.
- The receiving tag, which indicates the code set the server is using for international characters it transmits over the network. The receiving tag has the **out** parameter attribute and is applied to international character data declared in the operation’s output parameters. If the operation does not specify any international character output data, then it is not necessary to create this parameter.

You must define these code set tag parameters as unsigned long integers or unsigned long integers passed by reference. The receiving tag parameter must be declared as a pointer to the receiving tag unsigned long integer.

When international character data is to be unmarshalled, the client or server stub needs to have received a description of the code set being used before it receives the data. For this reason, the sending tag parameter must occur in an operation’s parameter list before any **in** international character data, and the receiving tag parameter must occur in an operation’s parameter list before any **out** international character data. The requirement that a tag must be received before the data it relates to is received also means that a customized binding handle cannot include international characters. This is because a binding handle must be the first parameter in a parameter list.

Here is an example **.idl** file for an interface named **cs_test** that uses the special international character type definition and the code set tag parameters for input and output parameters that are fixed arrays of characters from an international character set:

```
[
uuid(b076a320-4d8f-11cd-b453-08000925d3fe),
version(1.0)
]
interface cs_test
{
  const unsigned short SIZE = 100;
  typedef byte net_byte;

  error_status_t cs_fixed_trans (
[in] handle_t IDL_handle,
[in] unsigned long stag,
[in] unsigned long drtag,
[out] unsigned long *p_rtag,
[in] net_byte in_string[SIZE],
[out] net_byte out_string[SIZE]
  );
```

Writing the Attribute Configuration File

The next step in building an RPC application that supports character and code set interoperability is to create an attribute configuration file (**.acf**) to be associated with the **.idl** file. This **.acf** file uses the following attributes:

- The **cs_char** attribute, which associates the local data type that the application code uses to represent international characters in the local code set with the special **typedef** defined in the **.idl** file. This is a required ACF attribute for an RPC application that passes international character data. “Chapter 18. Interface Definition Language” on page 357 provides complete details on how to specify the **cs_char** ACF attribute and the programming restrictions associated with its use.
- The **cs_stag**, **cs_drtag**, and **cs_rtag** attributes, for each operation in the interface that specifies sending tag, desired receiving tag, and/or receiving tag parameters. These ACF attributes declare the tag parameters defined in the corresponding **.idl** file to be special code set tag parameters. Operations defined in the **.idl** file that specify international character in input parameters must use the **cs_stag** attribute. Operations defined in the **.idl** file that specify international character in output parameters must use the **cs_drtag** and **cs_rtag** attributes. “Chapter 19. Attribute Configuration Language” on page 425 provides complete details on how to specify the **cs_stag**, **cs_drtag**, and **cs_rtag** ACF attributes.
- The **cs_tag_rtn** attribute, which specifies the name of a routine that the client and server stubs will call to set an operation’s code set tag parameters to specific code set values. The **cs_tag_rtn** attribute is an optional ACF attribute for internationalized RPC applications; application developers can use it to provide code set tag transparency for callers of their application’s operations. See “Chapter 19. Attribute Configuration Language” on page 425 for complete details on how to specify the **cs_tag_rtn** attribute. “Tag-Setting Routine” on page 292 provides more information on the role of the tag-setting routine.

Here is the companion **.acf** file for the **cs_test** interface defined in “Writing the Interface Definition File” on page 286:

```
[
explicit_handle
]
interface cs_test
{
  include "dce/codesets_stub";

  typedef [cs_char(cs_byte)] net_byte;
  [comm_status, cs_tag_rtn(rpc_cs_get_tags)] cs_fixed_trans (
[cs_stag] stag,
[cs_drtag] drtag,
[cs_rtag] p_rtag);
```

The ACF for **cs_test** uses the **cs_char** attribute to define **net_byte** as a data type that represents international characters. Note that the local type specified in the **cs_char** attribute definition is **cs_byte**. This local type is analogous to the **byte** type. The ACF for **cs_test** also uses the **cs_tag_rtn** attribute to specify a tag-setting routine.

Writing the Stub Support Routines

When you use the **cs_char** attribute to define an international character data type, you must provide stub support routines that check the buffer storage requirements for character data to be converted and stub support routines that perform the conversions between the local and network code sets. And, if you use the **cs_tag_rtn** attribute, you must provide the routine that sets the code set tag parameters for the operations in the application that transfer international characters.

DCE RPC provides several buffer-sizing routines and one tag-setting routine. You can use the DCE RPC routines, or you can develop your own customized buffer-sizing and tag-setting routines; the choice depends upon your application's requirements. The next sections describe these types of stub support routines in more detail.

Buffer-Sizing Routines

Different code sets use different numbers of bytes to encode a single character. Consequently, there is always the possibility that the converted string can be larger than the original string when converting data from one code set to another. The function of the buffer-sizing routines is to calculate the necessary buffer size for code set conversion between local and network code sets and return their findings to the client and server stubs, which call these buffer-sizing routines before marshalling and unmarshalling any international character data. The stubs then allocate a new buffer, if necessary, before calling the code set conversion routines.

You must provide the following buffer-sizing routines for each local type that you define with the **cs_char** attribute:

- *local_type_name*_**net_size()**—Calculates the necessary buffer size for code set conversion from a local code set to a network code set. Client and server stubs call this routine before they marshall any international character data.
- *local_type_name*_**local_size()**—Calculates the necessary buffer size for code set conversion from a network code set to a local code set. Client and server stubs call this routine before they unmarshall any international character data.

You specify the name for the local data type in the *local_type_name* portion of the function name and the appropriate suffix name (**_net_size** or **_local_size**).

DCE RPC provides buffer-sizing routines for the **cs_byte** and **wchar_t** data types. The **cs_byte** data type is equivalent to the **byte** type, while the **wchar_t** data type is a platform-dependent data type whose range of values can represent encodings for all members of the largest international character set that exists within the set of character/code sets supported on the host.

The DCE RPC buffer-sizing routines are

- **cs_byte_net_size()**—Calculates the necessary buffer size for code set conversion from a local code set to a network code set when the **cs_byte** type has been specified as the local data type in the **.acf** file.
- **cs_byte_local_size()**—Calculates the necessary buffer size for code set conversion from a network code set to a local code set when the **cs_byte** type has been specified as the local data type in the **.acf** file.
- **wchar_t_net_size()**—Calculates the necessary buffer size for code set conversion from a local code set to a network code set when the **wchar_t** data type has been specified as the local data type in the **.acf** file.
- **wchar_t_local_size()**—Calculates the necessary buffer size for code set conversion from a network code set to a local code set when the **wchar_t** data type has been specified as the local data type in the **.acf** file.

If your internationalized RPC application uses either of these data types as the local type in the ACF, it can use these DCE RPC buffer-sizing routines; in order to do so, simply link with the DCE library when compiling your application. The example ACF shown earlier in this chapter uses the **cs_byte** type as the local type. Consequently, the client and server stubs will use the **cs_byte_** buffer-sizing routines. Refer to the **cs_byte_** ***(3rpc)** and **wchar_t_*****(3rpc)** reference pages for a description of the **cs_byte_** and **wchar_t_** routine signatures and functions.

Applications that use data types other than **cs_byte** or **wchar_t** as their local data types will need to provide their own buffer-sizing routines. User-provided buffer-sizing routines must follow the same signature as the DCE RPC-provided buffer-sizing routines. See the **cs_byte_** ***(3rpc)** and **wchar_t_*****(3rpc)** reference pages for a description of the required routine signatures.

Code Set Conversion Routines

When RPC clients and servers exchange international character data, the data being exchanged needs to be understood by both client and server. Both client and server need to understand a character set, and both client and server need to understand the way that character set is expressed. Code set conversion provides a way for a character set to be represented in a form that both client and server can understand, given that the client and server are using a compatible character set. (In general, character set conversion is not recommended; it is unlikely that clients and servers would want to map, for example, German characters to Chinese characters due to the data loss that would occur as a result.)

The stub support routines for code set conversion provide the mechanism for the stubs to use to convert between different code sets, given that character set compatibility has been established. The code set conversion routines translate a character set from one encoding to another. Consequently, the code set conversion routines provide the way for a character set to be represented in a form that both client and server can understand.

You must provide the following code set conversion routines for each local type that you define with the **cs_char** attribute:

- *local_type_name*_**to_netcs()**—Converts international character data from a local code set to a network code set. Client and server stubs call this routine before they marshall any international character data.
- *local_type_name*_**from_netcs()**—Converts international character data from a network code set to a local code set. Client and server stubs call this routine before they unmarshall any international character data.

You specify the name for the local data type in the *local_type_name* portion of the function name and the appropriate suffix name (**_to_netcs** or **_from_netcs**).

DCE RPC provides code set conversion routines for the **cs_byte** and **wchar_t** data types. These routines are

- **cs_byte_to_netcs()**—Converts international character data from a local code set to a network code set when the **cs_byte** type has been specified as the local data type in the **.acf** file.
- **cs_byte_from_netcs()**—Converts international character data from a network code set to a local code set when the **cs_byte** type has been specified as the local data type in the **.acf** file.
- **wchar_t_to_netcs()**—Converts international character data from a local code set to a network code set when the **wchar_t** data type has been specified as the local data type in the **.acf** file.
- **wchar_t_from_netcs()**—Converts international character data from a network code set to a local code set when the **wchar_t** data type has been specified as the local data type in the **.acf** file.

If your application uses either of these data types as the local type, it can use these DCE RPC code set conversion routines; in order to do so, simply link with the DCE library when compiling your application. Refer to the **cs_byte_ *(3rpc)** and **wchar_t_*(3rpc)** reference pages for a description of the **cs_byte_** and **wchar_t_** routine signatures and functions.

Applications that use data types other than **cs_byte** or **wchar_t** as their local data types will need to provide their own code set conversion routines. User-provided code set conversion routines must follow the same signature as the DCE RPC-provided code set conversion routines. See the **cs_byte_ *(3rpc)** and **wchar_t_ *(3rpc)** reference pages for a description of the **cs_byte_** and **wchar_t_** routine signatures and functions.

The DCE code set conversion routines depend upon the presence of the XPG4 **iconv** code set conversion facility in the underlying operating system platform. The **iconv** facility consists of the following routines:

- **iconv_open()**—Code conversion allocation function; returns a conversion descriptor that describes a conversion from the code set specified in one string pointer argument to the code set specified in another string pointer argument.
- **iconv()**—Code conversion function; converts the sequence of characters from one code set into a sequence of corresponding characters in another code set.
- **iconv_close()**—Code conversion deallocation function; deallocates the conversion descriptor and all associated resources allocated by the **iconv_open()** function.

Note that the **iconv** facility identifies a code set by a string name. This string name is the name that the local platform uses to refer to the code set. However, all of the stub support routines for automatic code set conversion use the unique identifier assigned to the code set in the code set registry to identify a code set. Before the DCE code set conversion routines can invoke the **iconv** facility, they must access the code set registry to retrieve the platform-specific string names associated with the local and network code set identifiers.

The DCE code set conversion routines use the **dce_cs_loc_to_rgy()** and **dce_cs_rgy_to_loc()** routines to access the code set registry and translate between code set string names and their corresponding unique identifiers. The *OSF*

DCE Application Development Reference provides a description of these routines' signatures and functions; developers who are writing their own code set conversion routines and who are using the **iconv** facility for conversion may want to use these DCE routines to convert between code set names and identifiers.

Tag-Setting Routine

Recall from "Writing the Interface Definition File" on page 286 that operations that specify international characters as input and output parameters declare special code set tag parameters. The purpose of these parameters is to hold the unique identifier for the code set into which the input or output data is to be encoded when it is transferred over the network.

The function of the tag-setting routine is to provide a way to set an operation's code set tag parameters to specific code set values from within the stubs rather than in the application code. The application specifies the name of the tag-setting routine as the argument to the **cs_tag_rtn** ACF attribute; the client and server stubs call this routine when the operation is invoked to set the tag parameters to specific network code set values before they call the stub support routines for buffer-sizing and code set conversion. The stubs use the network code set values returned by the tag-setting routine as input to the buffer-sizing and conversion routines. In turn, these routines compare the network code set values to be used for input and output data to the local code set in use for the data, and determine whether or not new buffer allocation and code set conversion are necessary.

When called from the client stub, the tag-setting routine sets the sending tag parameter to the code set to use for input character data. If the client expects output character data from the server, the routine also sets the desired receiving tag parameter to the code set that the client prefers the server to use for sending back the output data. On the client side, the buffer-sizing routines *local_type_name*_**net_size()** and the code set conversion routines *local_type_name*_**to_netcs()** use the value in the sending tag as the network code set value to use for transmitting the input data. When the input data arrives at the server side, the server stub uses the sending tag as input to the *local_type_name*_**local_size()** buffer-sizing routine and the *local_type_name*_**from_netcs()** code set conversion routines, which use the value to determine whether or not new buffer allocation and conversion is necessary from the network code set to the local code set.

When called from the server stub, the tag-setting routine sets the receiving tag parameter to the code set to use for transmitting the output character data back to the server. The routine can use the desired receiving tag value as input to determine the most appropriate code set in which to encode output data for the client. On the server side, the buffer-sizing routines *local_type_name*_**net_size()** and the code set conversion routines *local_type_name*_**to_netcs()** use the value in the receiving tag as the network code set value to use for transmitting the output data. When the output data arrives at the client side, the client stub uses the receiving tag as input to the *local_type_name*_**local_size()** buffer-sizing routine and the *local_type_name*_**from_netcs()** code set conversion routines, which use the value to determine whether or not new buffer allocation and conversion is necessary from the network code set to the local code set.

DCE RPC provides one tag-setting routine named **rpc_cs_get_tags()** that applications can use to set code set tag values within the stubs. To use this routine, specify its name as the argument to the **cs_tag_rtn** attribute and link your application with the DCE library. The example ACF for the **cs_test** interface specifies the **rpc_cs_get_tags()** routine.

Note that the **rpc_cs_get_tags()** routine always sets the receiving tag value on the server side to the value that the client specified in the desired receiving tag. See the **rpc_cs_get_tags(3rpc)** reference page for an explanation of this routine's signature and function.

RPC application programmers who are developing their own tag-setting routines can also refer to the **rpc_cs_get_tags(3rpc)** reference page to obtain the required signature for their user-written routine.

The tag-setting routine generally obtains the code set tag values from the binding handle. These values are usually determined by the character and code set evaluation routine invoked during the server binding import process, although they can be explicitly set in the binding handle by using the **rpc_cs_binding_set_tags()** routine. However, applications can design the tag-setting routine to perform evaluation within the stubs rather than in the application (client) code. For example, when called from the client side, the DCE RPC tag-setting routine **rpc_cs_get_tags()** performs character and code set compatibility evaluation itself if it does not find the tag values in the binding handle. See "Writing the Evaluation Routine" on page 303 for more information on deferred evaluation.

Writing the Server Code

A programmer who is developing an RPC server that supports character and code set interoperability needs to add the following steps to the server's initialization functions in addition to the normal initialization functions it carries out for RPC:

- Setting the server's locale
- Establishing the server's supported code sets
- Registering the server's supported code sets in the name service database
- Establishing a cleanup function that removes the server's supported code sets from the name service database on the server's termination or exit.

The next sections explain these steps in detail.

Setting the Server's Locale

The server initialization code needs to include a platform-specific routine that sets the locale environment for the server. This step establishes

- The name of the server's local code set.
- The names of the code sets for which converters exist on the host and consequently, into which processes that run on the host can convert if necessary.

An example of a locale-setting function is the POSIX, XPG3, XPG4 **setlocale()** function, which is defined in **locale.h**. The server code should call the locale-setting function as the first step in the initialization code, before calling the DCE RPC routines that register the interface and export the binding information.

The locale-setting function also establishes the value for two platform-specific macros that indicate

- The maximum number of bytes the local code set uses to encode one character.

- The maximum number of bytes that any of the supported code sets on the host will use to encode one character.

On POSIX, XPG3, and XPG4 platforms, these macros are **MB_CUR_MAX** and **MB_LEN_MAX** and are defined in **stdlib.h** and **limits.h**, respectively. The buffer-sizing routines use **MB_CUR_MAX** when calculating the size of a new buffer to hold converted character data.

Note that all hosts that are members of an internationalized DCE cell (that is, a cell that supports internationalized RPC applications) must provide converters that convert between their supported code sets and the ISO 10646 universal code set. The DCE RPC functions for character and code set interoperability use the universal code set as the default intermediate code set into which a client or server can convert if there are no other compatible code sets between them. “Writing the Evaluation Routine” on page 303 discusses intermediate code sets in more detail.

Establishing the Server’s Supported Code Sets

The next step in writing an internationalized RPC server is to add to the server’s initialization code a call to the DCE RPC routine **rpc_rgy_get_codesets()**. This routine gets the supported code set names defined in the locale environment and translates those names to their unique identifiers by accessing the code set registry on the host. The server initialization code should call this routine after it has registered the interface and created a server entry for its binding information in the name service database (by calling the DCE RPC NSI binding export routine **rpc_ns_binding_export()**).

The routine returns an array of unique identifiers from the code set registry that correspond to the server’s local code set and the code sets into which the server can convert, if necessary; this data structure is called the code sets array. The code sets array also contains, for each code set, the maximum number of bytes that code set uses to encode one character.

The purpose of this step is to obtain the registered unique identifiers for the server’s supported code sets for use by the DCE character and code set interoperability features, rather than using the string names for the code sets. The DCE features for character and code set interoperability do not use string names because different operating systems commonly use different string names to refer to the same code set, and clients and servers passing international characters in a cell of heterogeneous platforms need to ensure that they both refer to the same code set when establishing compatibility.

The code set registry provides the means for clients and servers to uniquely identify a code set while permitting different platforms and the code set converters offered on those platforms to continue to use the string names for the code sets.

See the **rpc_rgy_get_codesets(3rpc)** reference pages for a description of the **rpc_rgy_get_codesets()** routine’s signature and arguments.

Registering the Server’s Supported Code Sets in the Namespace

The next step in writing an internationalized RPC server is to make a call in the server’s initialization code to the DCE RPC routine **rpc_ns_mgmt_set_attribute()**, which takes the code sets array returned by **rpc_rgy_get_codesets()** and exports it to the server's entry in the name service database. The routine creates a code sets NSI attribute in the name service database and associates it with the server entry created by the NSI export operation.

The purpose of this step is to register the server's supported code sets into the name service database so that clients can gain access to the information. Note, then, that server entries for internationalized RPC servers will have code sets attributes in addition to the binding attributes and object attributes for the servers. For a general discussion of NSI attributes, see "Chapter 14. RPC and Other DCE Components" on page 195. Refer to the **rpc_ns_mgmt_set_attribute(3rpc)** reference page for a description of the **rpc_ns_mgmt_set_attribute()** routine's signature and arguments.

Establishing a Cleanup Function for the Namespace

The last step in writing an internationalized RPC server is to add a call to the DCE RPC routine **rpc_ns_mgmt_remove_attribute()** to the cleanup code within the server's initialization code. This DCE RPC routine will remove the code sets attribute associated with the server entry from the name service database when it is called from the cleanup code as the result of a server crash or exit. See the **rpc_ns_mgmt_remove_attribute(3rpc)** reference page for a description of the **rpc_ns_mgmt_remove_attribute()** routine's signature and arguments.

Sample Server Code

Here is an example of an internationalized RPC server that exports the **cs_test** interface defined in "Writing the Interface Definition File" on page 286.

```
#include <stdio.h>
#include <stdlib.h>
#include <dce/rpc.h>
#include <dce/nsattrid.h>
#include <dce/dce_error.h>
#include <locale.h>
#include <pthread.h>
#include <dce/codesets.h>
#include "cs_test.h"

/*
 * Macro for result checking
 */
#define CHECK_STATUS(t, func, returned_st, expected_st) \
{ \
  if (returned_st == expected_st) { \
  } \
  else { \
dce_error_inq_text(returned_st, \
 (unsigned char *)unexpected, &dce_status); \
dce_error_inq_text(expected_st,\
 (unsigned char *)expected, &dce_status); \
printf("FAILED %s()\nresult:  %s\nexpected:  %s\n\n", \
  func, unexpected, expected); \
  } \
} \

static unsigned char unexpected[dce_c_error_string_len];
static unsigned char expected[dce_c_error_string_len];
static int dce_status;

int
main(int argc, char *argv[])
{
  error_status_t status;
  int i;
  rpc_ns_handle_t inq_contxt;
  rpc_binding_vector_t *binding_vector;
  rpc_codeset_mgmt_p_t  arr;
  pthread_t this_thread = pthread_self();
  sigset_t sigset;
  char *nsi_entry_name;
  char *server_locale_name;
  error_status_t expected = rpc_s_ok;
  int server_pid;
  /* The environment variable I18N_SERVER_ENTRY needs
   * to be set before running this program. This is
   * not a DCE environment variable, so you can set up
   * your own environment variable if you like.
   */

  nsi_entry_name = getenv("I18N_SERVER_ENTRY");

  (void)pthread_mutex_init(&mutex, pthread_mutexattr_default);

  /* Set the locale. In this way, the current locale
   * information is extracted from XPG/POSIX defined
   * environment variable LANG or LC_ALL.
   */

  setlocale(LC_ALL, "");

  /*
   * Get supported code sets.
   */
  rpc_rgy_get_codesets (
&arr,
&status);

CHECK_STATUS(TRUE, "rpc_rgy_get_codesets", status, expected);

  rpc_server_register_if (
cs_test_v1_0_s_ifspec,
NULL,
NULL,
&status);

  CHECK_STATUS(TRUE, "rpc_server_register_if", status, expected);

  rpc_server_use_all_protseqs (
rpc_c_protseq_max_reqs_default,
&status);

  CHECK_STATUS(TRUE, "rpc_server_use_all_protseqs", status, expected);

  rpc_server_inq_bindings (
&binding_vector,
&status);

  CHECK_STATUS(TRUE, "rpc_server_inq_bindings", status, expected);
  rpc_ns_binding_export (
rpc_c_ns_syntax_default,
(unsigned_char_p_t)nsi_entry_name,
cs_test_v1_0_s_ifspec,
binding_vector,
NULL,
&status);

  CHECK_STATUS(TRUE, "rpc_ns_binding_export", status, expected);

  rpc_ep_register (
cs_test_v1_0_s_ifspec,
binding_vector,
NULL,
NULL,
&status);

  CHECK_STATUS(TRUE, "rpc_ep_register", status, expected);

  /*
   * Register the server's supported code sets into the name space.
   */
  rpc_ns_mgmt_set_attribute (
rpc_c_ns_syntax_default,
(unsigned_char_p_t)nsi_entry_name,
rpc_c_attr_codesets,
(void *)arr,
&status);

  CHECK_STATUS(TRUE, "rpc_ns_mgmt_set_attribute", status, expected);

  /*
   * Free memory allocated by getting code sets.
   */
  rpc_ns_mgmt_free_codesets (&arr, &status);

  CHECK_STATUS(TRUE, "rpc_ns_mgmt_free_codeset", status, expected);

  sigemptyset(&sigset);
  sigaddset(&sigset, SIGINT);

  if (pthread_signal_to_cancel_np(&sigset, &this_thread) != 0)
  {
printf("pthread_signal_to_cancel_np failed\n");
exit(1);
  }
  TRY
  {
server_pid = getpid();

printf("Listening for remote procedure calls...\n");

rpc_server_listen (
  rpc_c_listen_max_calls_default,
&status);

CHECK_STATUS(TRUE, "rpc_server_listen", status, expected);

/*
 * Remove code set attributes from namespace on return.
 */
rpc_ns_mgmt_remove_attribute (
  rpc_c_ns_syntax_default,
  (unsigned_char_p_t)nsi_entry_name,
  rpc_c_attr_codesets,
  &status);

CHECK_STATUS(TRUE, "rpc_ns_mgmt_remove_attribute", status, \
 expected);

rpc_ns_binding_unexport (
  rpc_c_ns_syntax_default,
  (unsigned_char_p_t)nsi_entry_name,
  cs_test_v1_0_s_ifspec,
  (uuid_vector_p_t)NULL,
  &status);

CHECK_STATUS(TRUE, "rpc_ns_binding_unexport", status, expected);
```

```
rpc_ep_unregister (
  cs_test_v1_0_s_ifspec,
  binding_vector,
  NULL,
  &status);

CHECK_STATUS(TRUE, "rpc_ep_unregister", status, expected);

rpc_binding_vector_free (
  &binding_vector,
  &status);

CHECK_STATUS(TRUE, "rpc_binding_vector_free", status, expected);

rpc_server_unregister_if (
  cs_test_v1_0_s_ifspec,
  NULL,
  &status);

CHECK_STATUS(TRUE, "rpc_server_unregister_if", status, expected);

(void)pthread_mutex_destroy(&mutex);
  }
  CATCH_ALL
  {
/*
 * Remove code set attribute from namespace on a signal.
 */
rpc_ns_mgmt_remove_attribute (
  rpc_c_ns_syntax_default,
  (unsigned_char_p_t)nsi_entry_name,
  rpc_c_attr_codesets,
  &status);

CHECK_STATUS(TRUE, "rpc_ns_mgmt_remove_attribute", status, \
 expected);

rpc_ns_binding_unexport (
  rpc_c_ns_syntax_default,
  (unsigned_char_p_t)nsi_entry_name,
  cs_test_v1_0_s_ifspec,
  (uuid_vector_p_t)NULL,
  &status);

CHECK_STATUS(TRUE, "rpc_ns_binding_unexport", status, expected);

rpc_ep_unregister (
  cs_test_v1_0_s_ifspec,
  binding_vector,
  NULL,
  &status);

CHECK_STATUS(TRUE, "rpc_ep_unregister", status, expected);

rpc_binding_vector_free (
  &binding_vector,
  &status);

CHECK_STATUS(TRUE, "rpc_binding_vector_free", status, expected);

rpc_server_unregister_if (
  cs_test_v1_0_s_ifspec,
  NULL,
  &status);

CHECK_STATUS(TRUE, "rpc_server_unregister_if", status, expected);
```

```
(void)pthread_mutex_destroy(&mutex);
  }
  ENDTRY;
}
```

Writing the Client Code

A programmer who is developing an RPC client that supports character and code set interoperability needs to add the following steps to the client code in addition to the basic functions for RPC:

1. Setting the client's locale
2. Establishing a character and code set compatibility evaluation routine that the NSI server binding import routines will call to evaluate potential servers for character and code set compatibility

The next sections explain these steps in detail.

Setting the Client's Locale

The first step in developing an internationalized RPC client is to add a call within the client code to a platform-specific function that sets the locale environment for the client. This step establishes

- The name of the client's local code set.
- The names of the code sets for which converters exist on the host and, consequently, into which processes that run on the host can convert if necessary.

The call to the locale-setting function must be the first call made within the client code. An example of a locale-setting function is the POSIX, XPG3, XPG4 **setlocale()** function, which is defined in **locale.h**.

The locale-setting function also establishes the value for two platform-specific macros that indicate

- The maximum number of bytes the local code set uses to encode one character.
- The maximum number of bytes that any of the supported code sets on the host will use to encode one character.

On the POSIX, XPG3, XPG4 platform, these macros are **MB_CUR_MAX** and **MB_LEN_MAX** and are defined in **stdlib.h** and **limits.h**, respectively. The buffer-sizing routines use the **MB_CUR_MAX** macro when calculating the size of a new buffer to hold converted character data.

Note that all hosts that are members of an internationalized DCE cell must provide converters that convert between their supported code sets and the ISO 10646 universal code set. The DCE RPC functions for character and code set interoperability use the universal code set as the default intermediate code set into which a client or server can convert if there are no other compatible code sets between them. "Writing the Evaluation Routine" on page 303 discusses intermediate code sets in more detail.

Establishing the Compatibility Evaluation Routine

The last step in writing an internationalized RPC client is to call the DCE RPC NSI routine **rpc_ns_import_ctx_add_eval()**. The purpose of this NSI routine is to add evaluation routines to the import context created by the

rpc_ns_binding_import_begin() routine that the NSI routine **rpc_ns_binding_import_next()** will call to perform additional compatibility evaluation on potential servers.

The internationalized RPC client code calls the **rpc_ns_import_ctx_add_eval()** routine to set up one or more character and code set compatibility evaluation routines to be called from **rpc_ns_binding_import_next()**. The client code must make the call to **rpc_ns_import_ctx_add_eval()** once for each compatibility routine that it wants to add to the import context for **rpc_ns_binding_import_next()**. See the **rpc_ns_import_ctx_add_eval(3rpc)** reference page for a description of the **rpc_ns_import_ctx_add_eval()** routine's signature and arguments.

The **rpc_ns_import_ctx_add_eval()** must be used in conjunction with the **rpc_ns_binding_import_begin/next/done()** suite of RPC NSI binding functions because these functions provide an import context argument. If you want to use the **rpc_ns_binding_lookup_begin/next/done/select()** suite of RPC NSI routines, your client code will need to perform character and code set evaluation logic on the binding handle returned by **rpc_ns_binding_select()**. "Example Character and Code Set Evaluation Logic" on page 306 provides a sample client that performs character and code set evaluation in conjunction with the **lookup** and **select** RPC NSI routines.

Sample Client Code

Here is an example of an internationalized RPC client that calls the operation defined in the **cs_test** interface shown in "Writing the Interface Definition File" on page 286 . The client establishes the DCE RPC evaluation routine **rpc_cs_eval_without_universal()** as the character and code set evaluation routine to use.

```
#include <stdio.h>
#include <locale.h>
#include <dce/rpc.h>
#include <dce/rpcsts.h>
#include <dce/dce_error.h>

#include "cs_test.h" /* IDL generated header file */

/*
 * Result check MACRO
 */
#define CHECK_STATUS(t, func, returned_st, expected_st) \
{ \
if (returned_st == expected_st) { \
/*
 * Do nothing.
 */
  } else { \
dce_error_inq_text(returned_st,\
 (unsigned char *)unexpected, &dce_status); \
dce_error_inq_text(expected_st, \
 (unsigned char *)expected, &dce_status); \
printf("FAILED %s()\nresult:  %s\nexpected:  %s\n\n", \
  func, unexpected, expected); \
  } \
} \

static unsigned char unexpected[dce_c_error_string_len];
static unsigned char expected[dce_c_error_string_len];
static int dce_status;
void
main(void)
```

```
{
  rpc_binding_handle_t  bind_handle;
  rpc_ns_handle_t     import_context;
  error_status_t   status;
  error_status_t   temp_status;
  cs_byte   net_string[SIZE];
  cs_byte   loc_string[SIZE];
  unsigned char   err_buf[256];
  char   *nsi_entry_name;
  char   *client_locale_name;
  int   i, rpc_num;
  FILE   *fp_in, *fp_out;

  /* The environment variable I18N_SERVER_ENTRY needs
   * to be set before running this program. This is
   * not a DCE environment variable, so you can set up
   * your own environment variable if you like.
   */

  nsi_entry_name = getenv("I18N_SERVER_ENTRY");

  setlocale(LC_ALL, "");

  rpc_ns_binding_import_begin (
rpc_c_ns_syntax_default,
(unsigned_char_p_t)nsi_entry_name,
cs_test_v1_0_c_ifspec,
NULL,
&import_context,
&status);

  CHECK_STATUS(TRUE, "rpc_ns_binding_import_begin", status, rpc_s_ok);

  /*
   * Add code set compatibility checking logic to the context.
   */
  rpc_ns_import_ctx_add_eval (
&import_context,
rpc_c_eval_type_codesets,
(void *)nsi_entry_name,
rpc_cs_eval_without_universal,
NULL,
&status);

  CHECK_STATUS(TRUE, "rpc_ns_import_ctx_add_eval", status, rpc_s_ok);
  while (1) {
rpc_ns_binding_import_next (
  import_context,
  &bind_handle,
  &status);

CHECK_STATUS(TRUE, "rpc_ns_binding_import_next", status, \
 rpc_s_ok);
  if (status == rpc_s_ok)
break;
  else
  {
return;
  }
  }

  rpc_ns_binding_import_done (
&import_context,
&status);

  CHECK_STATUS(TRUE, "rpc_ns_binding_import_done", status, rpc_s_ok);
  rpc_ep_resolve_binding (bind_handle,
cs_test_v1_0_c_ifspec,
&temp_status);

  CHECK_STATUS(TRUE, "rpc_ep_resolve_binding", temp_status, rpc_s_ok);

  if(rpc_mgmt_is_server_listening(bind_handle, &status)
&& temp_status == rpc_s_ok)
  {
; /* Do nothing. */
  }
  else
  {
dce_error_inq_text ((unsigned long)status,
 err_buf, (int *)&temp_status);
printf("is_server_listening error -> %s\n", err_buf);
  }
  /*
   * This program reads the data from a file.
   */

  fp_in = fopen("./i18n_input_data", "r");

  if (fp_in == NULL)
  {
printf("i18n_input_data open failed\n");
return;
  }

   fp_out = fopen("./i18n_method_fixed_result_file", "w");

   if (fp_out == NULL)
   {
printf("i18n_result_file open failed\n");
fclose(fp_in);
return;
  }

  rpc_num = 1;
  while (!feof(fp_in))
  {
(void)fgets((char *)net_string, SIZE, fp_in);

temp_status = cs_fixed_trans(bind_handle, net_string, loc_string);

if (temp_status != rpc_s_ok)
{
  dce_error_inq_text(temp_status, err_buf, (int *)&status);

  printf("FAILED %ld MSG: %s\n", (unsigned long)temp_status, \
   err_buf);
}
else
{
  printf("PASSED rpc #%d\n", rpc_num++);
  (void)fputs((char *)loc_string, fp_out);
  (void)fputs("\n", fp_out);
}
  }

  fclose(fp_in);
  fclose(fp_out);

  return;
}
```

Writing the Evaluation Routine

Recall from “Chapter 1. Introduction to DCE Facilities” on page 3 of the *OSF DCE Application Development Guide—Introduction and Style Guide* and “Chapter 11. Developing a Simple RPC Application” on page 149 of this guide that when a prospective client attempts to import binding information from a namespace entry that it looks up by name, the NSI import routine checks the binding for compatibility with the client by comparing interface UUIDs and protocol sequences. If the UUIDs match and the protocol sequences are compatible, the NSI operation considers the binding handle contained in the server entry to be compatible and returns it to the client. Internationalized clients need to perform additional compatibility checking on potential server bindings: they need to evaluate the server for character and code set compatibility.

The purpose of the character and code set compatibility evaluation routine is to determine

- Whether the character set the server supports is compatible with the client’s character set, since incompatible character sets result in unacceptable data loss during character conversion.
- The level of code set compatibility between client and server, which determines the conversion method that the client and server will use when transferring character data between them.

A conversion method is a process for converting one code set into another. There are four conversion methods:

- Receiver Makes It Right (RMIR)—The recipient of the data is responsible for converting the data from the sender’s code set to its own code set. This is the method that the RPC communications protocol uses to convert PCS character data between ASCII and EBCDIC code sets.
- Client Makes It Right (CMIR)—The client converts the input character data to be sent to the server into the server’s code set before the data is transmitted over the network, and converts output data received from the server from the server’s code set into its own local code set.
- Server Makes It Right (SMIR)—The server converts the input character data received from the client into its local code set from the client’s code set and converts output data to be sent to the client into the client’s code set before the data is transmitted over the network.
- Intermediate—Both client and server convert to a common code set. DCE defines a default intermediate code set to be used when there is no match between the client and server’s supported code sets; this code set is the ISO 10646 universal code set. Sites can also specify other code sets to be used as intermediate code sets in preference to ISO 10646; to do this, they run the **csrc** utility. See the **csrc(8dce)** reference pages for a description of **csrc** utility usage.

A character and code set compatibility evaluation routine generally employs a conversion model when determining the level of code set compatibility. A conversion model is an ordering of conversion methods; for example, CMIR first, then SMIR, then intermediate. Consequently, the actual conversion method used is determined at runtime.

DCE RPC Evaluation Routines

DCE RPC provides two character and code set compatibility evaluation routines: **rpc_cs_eval_with_universal()** and **rpc_cs_eval_without_universal()**. To use either one of these routines, specify their names in the evaluation function argument to the **rpc_ns_import_ctx_add_eval()** routine. (The sample client code shown in “Sample Client Code” on page 300 specifies a DCE RPC character and code set evaluation routine.)

The **rpc_cs_eval_with_universal()** routine first compares the client’s local code set with the server’s local code set. If they are the same, client-server character and code set compatibility exists. The routine returns to the NSI import routine, which returns the server binding to the client.

If the routine finds that the client and server local code sets differ, it calls the routine **rpc_cs_char_set_compat_check()** to determine client-server character set compatibility. If the client and server are using the same character set, it will be safe for them to exchange character data despite their use of different encodings for the character data. Clients and servers using different character sets are considered to be incompatible since the process of converting the character data from one character set to the other will result in significant data loss.

Using the client and server’s local code set identifiers as indexes into the code set registry, the **rpc_cs_char_set_compat_check()** routine obtains the registered values that represent the character set(s) that the specified code sets support. If the client and server support just one character set, the routine compares the values for compatibility. If the values do not match, then the client-server character sets are not compatible; for example, the client is using the German character set and the server is using the Korean character set. In this case, the routine returns the status code **rpc_s_ss_no_compat_charsets** to the evaluation routine so that binding to that server will be rejected.

If the client and server support multiple character sets, the **rpc_cs_char_set_compat_check()** routine determines whether at least two of the sets are compatible. If two or more sets match, the routine considers the character sets compatible and returns a success status code to the evaluation routine.

In the case where the client and server are character set compatible, the **rpc_cs_eval_with_universal()** routine uses the following model to determine a conversion method:

- RMIR (receiver makes it right)
- SMIR (client uses its local code set, server converts to and from it)
- CMIR (server uses its local code set, client converts to and from it)
- Use an intermediate code set
- Use the universal (ISO 10646) code set

This conversion model translates into the following steps:

1. The **rpc_cs_eval_with_universal()** routine takes the client’s local code set and searches through the server’s code sets array to determine whether it has a converter for the client’s local set. Then it takes the server’s local code set and searches through the client’s code sets array to see if it has a converter for the server’s local code set.
2. If both client and server support converters for each other’s local code sets (that is, they can convert to and from each other’s local code set), the routine sets the conversion method to RMIR.
3. If the server can convert to and from the client’s local code set, but the client cannot convert from the server’s local code set, the routine sets the conversion method to SMIR.

4. If the client can convert to and from the server's local code set, but the server cannot convert to and from the client's local code set, the routine sets the conversion method to CMIR.

   If the conversion method is SMIR or RMIR, the **rpc_cs_eval_with_universal()** routine sets both the sending tag and the desired receiving tag to the code set value that represents the client's local code set. In the case of CMIR, the routine sets both the sending tag and the desired receiving tag to the code set value that represents the server's local code set.
5. If neither client nor server support each other's local code set, the routine next determines if they both support a code set into which they both can convert to/from their local code sets. If it finds an intermediate set into which they both can convert, it sets the conversion method to INTERMEDIATE and sets the sending tag and desired receiving tag to the code set value that represents the intermediate code set to use.
6. If the routine does not find any intermediate code set into which client and server can convert, it sets the sending tag and desired receiving tag to the code set value that represents the ISO 10646 universal code set, which is the default intermediate code set that all DCE clients and servers support.

The **rpc_cs_eval_without_universal()** routine uses the following conversion model to determine a conversion method:
- RMIR
- SMIR (client uses its local code set, server converts to and from it)
- CMIR (server uses its local code set, client converts to and from it)
- Intermediate
- Reject for code set incompatibility

Consequently, the **rpc_cs_eval_without_universal()** uses the same evaluation logic as **rpc_cs_eval_with_universal()** except that it rejects the server binding if the client and server do not support a common code set to use as an intermediate code set.

Writing Customized Evaluation Routines

Programmers writing internationalized RPC applications can develop their own character and code set compatibility evaluation routines if their applications' needs are not met by the DCE RPC evaluation routines. These programmers may want to use the following DCE RPC routines within their evaluation routine:
- The **rpc_rgy_get_codesets()** routine
- The **rpc_cs_char_set_compat_check()** routine
- The **rpc_cs_binding_set_tags()** routine
- The **dce_cs_loc_to_rgy()** routine
- The **rpc_ns_mgmt_read_codesets()** routine
- The **rpc_ns_mgmt_free_codesets()** routine

Refer to the *OSF DCE Application Development Reference* for complete details about these routines.

Programmers who write their own evaluation routines can also select when evaluation is performed; that is, they can defer evaluation from occurring in the client code, or they can defer evaluation completely at the client side and let it take place in the server instead. Programmers who desire to defer evaluation to the client stub can write an evaluation routine that sets the client's and server's supported code sets into the binding handle returned by the client, then write the evaluation logic into the stub support routine for tag setting so that it performs evaluation within the client stub.

Applications that do evaluation in the client stub take the chance that the binding handle that is evaluated is the only binding handle available. For example, suppose there are three binding handles. Two are character and code set compatible, and one is incompatible. The incompatible binding is selected for RPC. If you evaluate in the tag-setting routine, you cannot reselect to get the other compatible bindings.

In general, it is recommended that character and code set evaluation take place in the client, rather than the server, for performance reasons. Also, once the server is selected and a connection is established between it and the client, the client cannot typically reselect the server because the code sets are incompatible.

Within the client, it is recommended that evaluation be performed in the client code rather than in the client stub because deferring evaluation to occur in the client stub removes any way for the client to gain access to other potential binding handles.

Notes About Tag Setting

The DCE RPC character and code set compatibility evaluation routines set the method and the code set tag values into a data structure in the binding handle returned to the client. These routines always set the sending tag and desired receiving tag to the same code set value.

In addition, if the application uses the DCE RPC routine **rpc_cs_get_tags()** to set the code set tags for the stubs, the value of the server's receiving tag will always be the value of what the client sent to it in the desired receiving tag. If RMIR is used, the desired receiving tag is the server's current code set.

RPC application programmers who do not want to use the DCE RPC-provided evaluation routines can use the **rpc_cs_binding_set_tags()** routine to set the code set tag values into a binding handle.

Example Character and Code Set Evaluation Logic

Here is an example client program of the **cs_test** interface that provides its own character and code set evaluation logic. This example client uses the **rpc_cs_binding_set_tags()** routine to set the code set tags within the client code rather than using a tag-setting routine to set them within the stub code.

```
#include <stdio.h>
#include <locale.h>
#include <dce/rpc.h>
#include <dce/rpcsts.h>
#include <dce/dce_error.h>

#include "cs_test.h" /* IDL generated header file */
/*
 * Result check MACRO
 */
#define CHECK_STATUS(t, func, returned_st, expected_st) \
{ \
  if (returned_st == expected_st) { \
  ; /* No operation */

  } else { \
dce_error_inq_text(returned_st,\
 (unsigned char *)unexpected, &dce_status); \
dce_error_inq_text(expected_st,\
 (unsigned char *)expected, &dce_status); \
printf("FAILED %s()\nresult:  %s\nexpected:  %s\n\n", \
  func, unexpected, expected); \
  } \
} \

static unsigned char  unexpected[dce_c_error_string_len];
static unsigned char  expected[dce_c_error_string_len];
static int       dce_status;


void
main(void)
{
  rpc_binding_handle_t bind_handle;
  rpc_ns_handle_t lookup_context;
  rpc_binding_vector_p_t bind_vec_p;
  unsigned_char_t *entry_name;
  unsigned32 binding_count;
  cs_byte net_string[SIZE];
  cs_byte loc_string[SIZE];
  int i, k, rpc_num;
  int model_found, smir_true, cmir_true;
  rpc_codeset_mgmt_p_t client, server;
  unsigned32 stag;
  unsigned32 drtag;
  unsigned16 stag_max_bytes;
  error_status_t status;
  error_status_t temp_status;
  unsigned char err_buf[256];
  char *nsi_entry_name;
  char *client_locale_name;
  FILE *fp_in, *fp_out;

  nsi_entry_name = getenv("I18N_SERVER_ENTRY");

  setlocale(LC_ALL, "");
  rpc_ns_binding_lookup_begin (
rpc_c_ns_syntax_default,
(unsigned_char_p_t)nsi_entry_name,
cs_test_v1_0_c_ifspec,
NULL,
rpc_c_binding_max_count_default,
&lookup_context,
&status);

  CHECK_STATUS(TRUE, "rpc_ns_binding_lookup_begin", status, rpc_s_ok);

  rpc_ns_binding_lookup_next (
lookup_context,
&bind_vec_p,
&status);

  CHECK_STATUS(TRUE, "rpc_ns_binding_lookup_next", status, rpc_s_ok);

  rpc_ns_binding_lookup_done (
&lookup_context,
&status);

  CHECK_STATUS(TRUE, "rpc_ns_binding_lookup_done", status, rpc_s_ok);

  /*
   * Get the client's supported code sets
   */
  rpc_rgy_get_codesets (
&client,
&status);

  CHECK_STATUS(TRUE, "rpc_rgy_get_codesets", status, rpc_s_ok);

  binding_count = (bind_vec_p)->count;
  for (i=0; i < binding_count; i++)
  {
if ((bind_vec_p)->binding_h[i] == NULL)
  continue;

rpc_ns_binding_select (
  bind_vec_p,
  &bind_handle,
  &status);

CHECK_STATUS(FALSE, "rpc_ns_binding_select", status, rpc_s_ok);
if (status != rpc_s_ok)
{
  rpc_ns_mgmt_free_codesets(&client, &status);
  CHECK_STATUS(TRUE, "rpc_ns_mgmt_free_codesets",
   status, rpc_s_ok);
}

rpc_ns_binding_inq_entry_name (
  bind_handle,
  rpc_c_ns_syntax_default,
  &entry_name,
  &status);

CHECK_STATUS(TRUE, "rpc_ns_binding_inq_entry_name", status, \
 rpc_s_ok);
if (status != rpc_s_ok)
{
  rpc_ns_mgmt_free_codesets(&client, &status);
  CHECK_STATUS(TRUE, "rpc_ns_mgmt_free_codesets",
   status, rpc_s_ok);
}

/*
    * Get the server's supported code sets from NSI
    */
rpc_ns_mgmt_read_codesets (
  rpc_c_ns_syntax_default,
  entry_name,
  &server,
  &status);

CHECK_STATUS(FALSE, "rpc_ns_mgmt_read_codesets", status, \
 rpc_s_ok);

if (status != rpc_s_ok)
{
rpc_ns_mgmt_free_codesets(&client, &status);
  CHECK_STATUS(TRUE, "rpc_ns_mgmt_free_codesets",
   status, rpc_s_ok);
}
/*
    * Start evaluation
    */
if (client->codesets[0].c_set == server->codesets[0].c_set)
{
  /*
   * client and server are using the same code set
   */
  stag = client->codesets[0].c_set;
  drtag = server->codesets[0].c_set;
  break;
```

```
}

/*
 * check character set compatibility first
 */
rpc_cs_char_set_compat_check (
  client->codesets[0].c_set,
  server->codesets[0].c_set,
  &status);

CHECK_STATUS(FALSE, "rpc_cs_char_set_compat_check",
 status, rpc_s_ok);

if (status != rpc_s_ok)
{
  rpc_ns_mgmt_free_codesets(&server, &status);
  CHECK_STATUS(TRUE, "rpc_ns_mgmt_free_codesets",
   status, rpc_s_ok);
}

smir_true = cmir_true = model_found = 0;

for (k = 1; k <= server->count; k++)
{
  if (model_found)
break;

  if (client->codesets[0].c_set
== server->codesets[k].c_set)
  {
smir_true = 1;
model_found = 1;
  }
  if (server->codesets[0].c_set
== client->codesets[k].c_set)
  {
cmir_true = 1;
model_found = 1;
  }
}

if (model_found)
{
  if (smir_true && cmir_true)
  {
/* RMIR model works */
stag = client->codesets[0].c_set;
drtag = server->codesets[0].c_set;
stag_max_bytes
  = client->codesets[0].c_max_bytes;
  }
  else if (smir_true)
  {
/* SMIR model */
stag = client->codesets[0].c_set;
drtag = client->codesets[0].c_set;
stag_max_bytes
  = client->codesets[0].c_max_bytes;
  }
  else
  {
/* CMIR model */
stag = server->codesets[0].c_set;
drtag = server->codesets[0].c_set;
stag_max_bytes
  = server->codesets[0].c_max_bytes;
  }
  /*
   * set tags value to the binding
   */
  rpc_cs_binding_set_tags (
&bind_handle,
stag,
drtag,
stag_max_bytes,
&status);
  CHECK_STATUS(FALSE, "rpc_cs_binding_set_tags",
   status, rpc_s_ok);
  if (status != rpc_s_ok)
  {
rpc_ns_mgmt_free_codesets(&server, &status);
CHECK_STATUS(FALSE, "rpc_ns_mgmt_free_codesets",
 status, rpc_s_ok);
rpc_ns_mgmt_free_codesets(&client, &status);
CHECK_STATUS(TRUE, "rpc_ns_mgmt_free_codesets",
 status, rpc_s_ok);
  }
}
else
{
  /*
   * try another binding
   */
  rpc_binding_free (
&bind_handle,
&status);

  CHECK_STATUS(FALSE, "rpc_binding_free", status, rpc_s_ok);
  if (status != rpc_s_ok)
  {
rpc_ns_mgmt_free_codesets(&server, &status);
CHECK_STATUS(FALSE, "rpc_ns_mgmt_free_codesets", \
 status, rpc_s_ok);
rpc_ns_mgmt_free_codesets(&client, &status);
CHECK_STATUS(TRUE, "rpc_ns_mgmt_free_codesets", \
 status, rpc_s_ok);
  }
}
  }

  rpc_ns_mgmt_free_codesets(&server, &status);
  CHECK_STATUS(FALSE, "rpc_ns_mgmt_free_codesets", status, rpc_s_ok);

  rpc_ns_mgmt_free_codesets(&client, &status);
  CHECK_STATUS(TRUE, "rpc_ns_mgmt_free_codesets", status, rpc_s_ok);

  if (!model_found)
  {
printf("FAILED No compatible server found\n");
tet_result(TET_DCE_FAIL);
  }

  rpc_ep_resolve_binding (bind_handle,
cs_test_v1_0_c_ifspec,
&temp_status);

  CHECK_STATUS(TRUE, "rpc_ep_resolve_binding", temp_status, rpc_s_ok);
  if(rpc_mgmt_is_server_listening(bind_handle, &status)
&& temp_status == rpc_s_ok)
  {
printf("PASSED rpc_mgmt_is_server_listening());
  }
  else
  {
dce_error_inq_text ((unsigned long)status, err_buf,
 (int *)&temp_status);
  printf("is_server_listening error -> %s\n", err_buf);
  }

  fp_in = fopen("./i18n_input_data", "r");

  if (fp_in == NULL)
  {
printf("i18n_input_data open failed\n");
tet_result(TET_DCE_FAIL);
  }

  fp_out = fopen("./i18n_tags_fixed_result_file", "w");

  if (fp_out == NULL)
  {
printf("i18n_result_file open failed\n");
tet_result(TET_DCE_FAIL);
  }

  rpc_num = 1;
  while (!feof(fp_in))
  {
(void)fgets((char *)net_string, SIZE, fp_in);

temp_status = cs_fixed_trans(bind_handle, net_string, loc_string);

if (temp_status != rpc_s_ok)
{
  dce_error_inq_text(temp_status, err_buf, (int *)&status);

  printf("FAILED %ld MSG: %s\n", (unsigned long)temp_status, \
   err_buf);
}
else
{
  printf("PASSED rpc #%d\n", rpc_num++);
  (void)fputs((char *)loc_string, fp_out);
  (void)fputs(", fp_out);
}
  }
  fclose(fp_in);
  fclose(fp_out);

  return;
}
```

Chapter 17. Topics in RPC Application Development

This chapter describes special features of DCE RPC for application development. The topics include

- Memory management
- Error handling
- Context handles
- Pipes
- Nested calls and callbacks
- Routing RPCs
- Portable data and the IDL encoding services

Memory Management

When called to handle a remote operation, RPC client stubs allocate and free memory by using whatever memory management scheme is currently in effect. The client code—the generic code that can be called from either RPC clients or RPC servers—can use DCE RPC stub support routines to control which memory management scheme the stubs will use.

If client code has not explicitly set the memory management routines, the RPC client stubs use the following defaults:

- When called from manager code, and the operation contains one or more parameters that are full or unique pointers, or the ACF **enable_allocate** attribute has been applied, the client stubs use the **rpc_ss_allocate()** and **rpc_ss_free()** routines.
- When called from any other context, the RPC client stubs use the operating system allocation and free routines (for example, **malloc()** and **free()**) on POSIX platforms.

Note that the memory management scheme established, whether explicitly or by default, is on a per-thread basis.

RPC server stubs do not allocate memory. Instead, they rely on the manager code—the code that the server stubs call—to allocate it for them.

The following sections gives guidelines for how client code and manager code should use the the various allocation and free routines provided with DCE.

Note: DCE provides two versions of DCE RPC stub support routines. The **rpc_ss_ *()** routines raise an exception, while the **rpc_sm_ *()** routines return an error status value. In all other ways, the routines are identical. It is generally recommended that you use the **rpc_sm_ *()** routines instead of the **rpc_ss_ *()** routines for compliance with the *Application Environment Specification/Distributed Computing*.

Using the Memory Management Defaults

If it does not matter to the client code which memory allocation routine the RPC client stubs use, the client code should call the **rpc_ss_client_free()** routine to free any memory that the client stub allocates and returns. The **rpc_ss_client_free()** routine uses the current free routine that is in effect. Client code that uses

rpc_ss_client_free() must use caution if it calls other routines before it frees all of the pieces of allocated storage with **rpc_ss_client_free()**, because it is possible that the called code has been written so that it swaps in a different allocation/free pair without reestablishing the previous allocation/free pair on exit.

Using rpc_ss_allocate and rpc_ss_free

Both client code and manager code can use **rpc_ss_allocate()** and **rpc_ss_free()**. The next sections describe how.

Using rpc_ss_allocate and rpc_ss_free in Manager Code

Manager code uses either the **rpc_ss_allocate()** and **rpc_ss_free()** routines or the operating system allocation and free routines to allocate and free memory.

Manager code uses **rpc_ss_allocate()** to allocate storage for data that the server stub is to send back to the client. Manager code can either use **rpc_ss_free()** to free the storage explicitly, or it can rely on the server stub to free it. After the server stub marshalls the output parameters, it releases any storage that the manager code has allocated with **rpc_ss_allocate()**.

Manager code can also use the **rpc_ss_free()** routine to release storage pointed to by a full pointer in an input parameter and have the freeing of the memory reflected on return to the calling application if the **reflect_deletions** attribute has been specified as an operation attribute in the interface definition. See “Chapter 18. Interface Definition Language” on page 357 for instructions on how to declare the **reflect_deletions** operation attribute.

Manager code uses the operating system allocation routine to create storage for its internal data. The server stub does not automatically free memory that operating system allocation routines have allocated. Instead, manager code must use the operating system free routine to deallocate the memory explicitly before it exits.

When manager code makes a remote call, the default memory management routines are **rpc_ss_allocate()** and **rpc_ss_free()**.

Using rpc_ss_allocate and rpc_ss_free in Client Code

Client code may also want to use the **rpc_ss_allocate()** and **rpc_ss_free()** routines as the stub memory management scheme. However, before client code can use **rpc_ss_allocate()** and **rpc_ss_free()**, it must first call the **rpc_ss_enable_allocate()** routine, which enables the use of **rpc_ss_allocate()**. If client code calls **rpc_ss_enable_allocate()**, it must also call the **rpc_ss_disable_allocate()** routine before it exits its thread to disable use of **rpc_ss_allocate()**. This routine releases all of the memory allocated by calls to **rpc_ss_allocate()** in that thread since the call to **rpc_ss_enable_allocate()** was made. As a result, client code can either free each piece of allocated storage with **rpc_ss_free()**, or it can have **rpc_ss_disable_allocate()** free it all at once when it disables the **rpc_ss_allocate/free** memory management scheme.

Before calling **rpc_ss_enable_allocate()**, client code must ensure that it has not been called by code that has already set up the **rpc_ss_allocate/free** memory management scheme. As a result, if the client code can ensure that it has not been called from a manager routine, *and* it can ensure that any previous calls to

rpc_ss_enable_allocate() have been paired with calls to **rpc_ss_disable_allocate()**, it can safely call **rpc_ss_enable_allocate()**.

If client code cannot ensure that these conditions are true, it should check to make sure the **rpc_ss_allocate/free** scheme has not already been set up. For example:

```
/* Get RPC memory allocation thread handle */

  rpc_ss_thread_handle_t thread_handle;
  idl_void_p_t (*p_saved_alloc)(unsigned long);
  void (*p_saved_free)(idl_void_p_t);

  TRY
    thread_handle = rpc_ss_get_thread_handle();
  CATCH(pthread_badparam_e)
    thread_handle = NULL;
  ENDTRY

  if (thread_handle == NULL)  {

   /* Set up rpc_ss_allocate environment */

    rpc_ss_enable_allocate();
  }

  rpc_ss_swap_client_alloc_free(
    appl_client_alloc,appl_client_free,
    &p_saved_alloc,&p_saved_free);
```

After control returns from the client stub, the client code should again check to see whether **rpc_ss_allocate/free** has already been enabled before it calls **rpc_ss_disable_allocate()**:

```
  rpc_ss_set_client_alloc_free(p_saved_alloc,p_saved_free);

/* If we set up rpc_ss_allocate environment, disable it now */

  if (thread_handle == NULL)

rpc_ss_disable_allocate();
```

Using Your Own Allocation and Free Routines

At times it might be necessary for client code to change the routines that the client stubs use to allocate and free memory. For example, client code that is making an RPC call might want to direct the RPC client stubs to use special debug versions of **malloc()** and **free()** that check for memory leaks. Another example might be an application that uses DCE RPC but needs to preserve its users' ability to free memory returned from the application by using the platform's memory management scheme (rather than exposing the user to DCE).

Client code that wants to use its own memory allocation and free routines can use the **rpc_ss_swap_client_alloc_free()** routine to exchange the current client allocation and freeing mechanism for one supplied in the call. The routine returns pointers to the memory allocation and free routines formerly in use. Before calling **rpc_ss_swap_client_alloc_free()**, client code must ensure that it has not been called from a manager routine.

Deallocation of allocated storage returned from the client stubs is not automatic. Therefore, client code must ensure that it uses the free routine that it specified in the call to **rpc_ss_swap_client_alloc_free()** to deallocate each piece of allocated storage.

Client code that swaps in memory management routines with **rpc_ss_swap_client_alloc_free()** should use the **rpc_ss_set_client_alloc_free()** routine before it exits to restore the old allocation and free routines.

Using Thread Handles in Memory Management

There are two situations where control of memory management requires the use of thread handles. The more common situation is when the manager thread spawns additional threads. The less common situation is when a program transitions from being a client to being a server, then reverts to being a client.

Spawning Threads

When a remote procedure call invokes the manager code, the manager code may wish to spawn additional threads to complete the task for which it was called. To spawn additional threads that are able to perform memory management, the manager code must first call the **rpc_ss_get_thread_handle()** routine to get its thread handle and then pass that thread handle to each spawned thread. Each spawned thread must call the **rpc_ss_set_thread_handle()** routine with the handle received from the manager code.

These routine calls allow the manager and its spawned threads to share a common memory management environment. This common environment enables memory allocated by the spawned threads to be used in returned parameters and causes all allocations in the common memory management environment to be released when the manager thread returns to the server stub.

The main manager thread must not return control to the server stub before all the threads it spawned complete execution; otherwise, unpredictable results may occur.

The listener thread can cancel the main manager thread if the remote procedure call is orphaned or if a cancellation occurs on the client side of the application. You should code the main manager thread to terminate any spawned threads before it exits. The code should anticipate exits caused by an unexpected exception or by being canceled.

Your code can handle all of these cases by including a **TRY/FINALLY** block to clean up any spawned threads if a cancellation or other exception occurs. If unexpected exceptions do not concern you, then your code can perform two steps. They are disabling cancelability before threads are spawned followed by enabling cancelability after the join operation finishes and after testing for any pending cancel operations. Following this disable/enable sequence prevents routine **pthread_join()** from producing a cancel point in a manager thread that has spawned threads which, in turn, share thread handles with the manager thread.

Transitioning from Client to Server to Client

Immediately before the program changes from a client to a server, it must obtain a handle on its environment as a client by calling **rpc_ss_get_thread_handle()**.

When it reverts from a server to a client, it must reestablish the client environment by calling the **rpc_ss_set_thread_handle()** routine, supplying the previously obtained handle as a parameter.

Guidelines for Error Handling

During a remote procedure call, server and communications errors may occur. These errors can be handled using any or all of the following methods:

- Writing exception handler code to recover from the error or to exit the application
- Using the **fault_status** attribute in the ACF to report an RPC server failure
- Using the **comm_status** attribute in the ACF to report a communications failure

Use of exceptions, where the procedure exits the program due to an error, tends to improve code quality. It does this by making errors obvious because the program exits at that point, and by lessening the amount of code needed to detect error conditions and handle them. When you use the **fault_status** attribute, an exception that occurs on the server is not reported to the client as an exception. The variable to which the **comm_status** attribute is attached contains error codes that report errors that would not have occurred if the application were not distributed over a communications network. The **comm_status** attribute provides a method of handling RPC errors without using an exception handler.

Exceptions

Exceptions report either RPC errors or errors in application code. Exceptions have the following characteristics:

- You do not have to adjust procedure declarations between local and distributed code.
- You can distribute existing interfaces without changing code.
- You do not have to check for failures. This results in more robust code because errors are reported even if they are not checked.
- Your code is more efficient when there is no recovery coded for failures.
- You can use a simpler coding style.
- Exceptions work well for coarse-grained exception handling.
- If your application does not contain any exception handlers and the application thread gets an error, the application thread is terminated and a system-dependent error message from the threads package is printed.

Note: RPC exceptions are equivalent to RPC status codes. To identify the status code that corresponds to a given exception, replace the **_x_** string of the exception with the string **_s_**. For example, the exception **rpc_x_comm_failure** is equivalent to the status code **rpc_s_comm_failure**. The RPC exceptions are defined in the **dce/rpcexc.h** header file, and the equivalent status codes are described in the *OSF DCE Problem Determination Guide* .

The set of exceptions that can always be returned from the server to the client (such as the **rpc_x_invalid_tag** exception) are referred to as *system exceptions*. These exceptions are defined in **dce/rpcexec.h** and **dce/exec_handling.h**.

An interface definition can also specify a set of user-defined exceptions that the interface's operations can return to the client. You can declare user-defined exceptions in an interface definition by using the **exceptions** interface attribute, which is described in “Chapter 18. Interface Definition Language” on page 357.

If a user-defined exception in the implementation of a server operation occurs during server execution, the server terminates the operation and propagates the exception to the client in a manner similar to the way system exceptions are propagated. If a server implementation of an operation raises an exception that is neither a system exception nor a user-defined exception, the exception returned to the client is **rpc_x_unknown_remote_fault**.

By default, the IDL compiler defines and initializes all exceptions under a once block in the generated stubs. If you want to share exception names in multiple interfaces or you desire greater control over how these exceptions are defined and initialized, you can use the ACF **extern_exceptions** attribute to disable the automated mechanism that the IDL compiler uses to define and initialize exceptions. See “Chapter 19. Attribute Configuration Language” on page 425 for more information on the **extern_exceptions** attribute.

Because exceptions are associated with operation implementation, they are not imported into other interfaces by way of the **import** declaration. For more information about using exceptions to handle errors, see “Part 2. DCE Threads” on page 101 of this guide.

The fault_status Attribute

The **fault_status** attribute requests that errors occurring on the server due to incorrectly specified parameter values, resource constraints, or coding errors be reported by a designated status parameter instead of by an exception.

If a user-defined exception is returned from a server to a client that has specified **fault_status** on the operation in which the exception occurred, the value given to the **fault_status** parameter is **rpc_s_fault_user_defined**.

The **fault_status** attribute has the following characteristics:

- Occurs where you do not want transparent local/remote behavior
- Occurs where you expect that you may be passing incorrect data to the server or the server is not coded robustly, or both
- Works well for fine-grained error handling
- Requires that you adjust procedure declarations between local and distributed code
- Controls the reporting only of errors that come from the server and that are reported via a fault packet

For more information on the **fault_status** attribute, see “Chapter 19. Attribute Configuration Language” on page 425.

The comm_status Attribute

The **comm_status** attribute requests that RPC communications failures be reported through a designated status parameter instead of by an exception. The **comm_status** attribute has the following characteristics:

- Occurs where you expect communications to fail routinely; for instance, no server is available, the server has no resources, and so on

- Works well for fine-grained error handling; for example, trying a procedure many times until it succeeds
- Requires that you adjust procedure declarations between local and distributed code to add the new status parameter
- Controls the reporting of errors only from RPC runtime error status codes

For more information on the **comm_status** attribute, see "Chapter 19. Attribute Configuration Language" on page 425.

Determining Which Method to Use for Handling Exceptions

Some conditions are better for using the **comm_status** or **fault_status** attribute on an operation, rather than the default approach of handling exceptions.

The **comm_status** attribute is useful only if the call to the operation has a specific recovery action to perform for one or more communications failures; for example, **rpc_s_comm_failure** or **rpc_s_no_more_bindings**. The **comm_status** attribute is recommended only when the application knows that it is calling a remote operation. If you expect communications to fail often because the server does not have enough resources to execute the call, you can use this attribute to allow the call to be retried several times. If you are using an implicit or explicit binding, you can use the **comm_status** attribute if you want to try another server because the operation cannot be performed on the one you are currently using. You can also use an exception handler for each of the two previous instances.

In general, the advantange of using **comm_status** if the recovery is local to the routine is that the overhead is less. The disadvantage of using **comm_status** is that it results in two different operation signatures. Distributed calls contain the **comm_status** attribute, however; local calls do not. Also, if all of the recovery cannot be done locally (where the call is made), there must be a way to pass the status to outer layers of code to process it.

The **fault_status** attribute is useful only if the call to the operation has a specific recovery action to perform for one or more server faults; for example, **rpc_s_invalid_tag**, **rpc_s_fault_pipe_comm_error**, **rpc_s_fault_int_overflow**, or **rpc_s_fault_remote_no_memory**. Use **fault_status** only when the application calls a remote operation and wants different behavior than if it calls the same operation locally. If you are requesting an operation on a large data set, you can use this attribute to trap **rpc_s_fault_remote_no_memory** and retry the operation to a different server, or you may break your data set into two smaller sections. You can also handle the previous case with exception handlers. The advantange of using **fault_status** if the recovery is local is that the overhead is less. The disadvantage of **fault_status** is that the operation is different between the local and distributed case. Also, if all of the recovery cannot be done locally, there must be a way to pass the status to outer layers of code to process it.

Examples of Error Handling

The following subsections present two examples of error handling. The first example assumes that the **comm_status** attribute is in use in the ACF. The second example assumes that the **comm_status** attribute is not in use.

The Matrix Math Server Example

Assume that you have an existing local interface that provides matrix math operations. Since it is local, errors such as floating-point overflow or divide by zero are returned to the caller of a matrix operation as exceptions. It is likely that these exceptions are caused by providing data to the operation in an improper form.

In this case, the exceptions are part of the interface, so **fault_status** changes the way the application calls the matrix interface and probably is undesirable. Depending on the environment, finding a server may not be difficult (if the network is relatively stable and has enough resources), and adding **comm_status** serves only to introduce differences between the local and distributed applications.

If a decision as to what action to take is based upon a communications failure, then you may try to add the conditional code **comm_status** requires. Otherwise, using **auto_handle** allows an attempt on each available server. If no server is available, the application terminates because it cannot proceed. You can add an exception handler to the main program to report the error in a user-friendly manner.

The Stock Quote Application Example

Assume that you have an application that reads from stock quote servers and displays graphs of the data. Since you do not expect to get server failures because it is a commercial-quality server, you are not interested in writing code to handle values returned from **fault_status**. If high availability and robustness is important, you may have a list of recovery plans to make sure a stock analyst can get the necessary information as quickly as possible. For example:

```
retry_count = 10;
do {
  query_stock_quote(h, ...,&st);
  switch (st)     /* st parameter can be used because */
  {          /* [comm_status] is in the ACF */
    case rpc_s_ok:
      break;
    case rpc_s_comm_failure:
      retry_count -= 1;
      break;
    case rpc_s_network_unreachable:
      h = some_other_handle;
      break;
    case
      .
      .
      .
    default:
      retry_count -= 1;
  }
}
while ((st == rpc_s_ok) || (retry_count <= 0))
```

If this is not a critical application, you may only report that the server is currently unavailable. Depending upon the design of the application, there may be several places to put the exception handler to report the failure but continue processing. For example:

```
TRY
  update_a_quote(...);
CATCH_ALL
  display_message("Stock quote not currently available");
ENDTRY
```

This example assumes that **update_a_quote()** eventually calls the remote operation **query_stock_quote()** and that this call may raise an exception that is detected and reported here.

The advantage of using exceptions in this case is that all of the work done in **update_a_quote()** has the same error recovery and it does not need to be repeated at every call to a remote operation. Another advantage is that, if one of the remote operations does have a recovery for one exception, it can handle that one exception and allow the rest to propagate to the more general handler in an outer layer of the code.

Context Handles

During a series of remote procedure calls, the client may need to refer to a context maintained by a specific server instance. Server application code can maintain information it needs for a particular client (such as the state of RPC the client is using) as a context. To provide a client with a means of referring to its context, the client and server pass back and forth an RPC-specific parameter called a context handle. A context handle is a reference (a pointer) to the server instance and the context of a particular client. A context handle ensures that subsequent remote procedure calls from the client can reach the server instance that is maintaining context for the client.

On completing the first procedure in a series, the server passes a context handle to the client. The context handle identifies the context that the server uses for subsequent operations. The client is not supposed to do anything with the context handle; it merely passes it to subsequent calls as needed, and it is used internally by the remote calls. This allows applications to have such things as remote calls that handle file operations much as local calls would; that is, a client application can remotely open a file, get back a handle to it, and then perform various other remote operations on it, passing the context handle as an argument to the calls. A context handle can be used across interfaces (where a single server offers the multiple interfaces), but *a context handle belongs only to the client that caused it to be activated*.

The server maintains the context for a client until the client calls a remote procedure that terminates use of the context or communications are lost. In the latter case, the server's runtime can invoke a context rundown procedure. This application-specific routine is called by the server stub automatically to reclaim (rundown) the pointed-to resource in the event of a communications break between the server and client. For example, in the case of the remote file pointer just mentioned, the context rundown routine would simply close the file.

As usual with RPC, you need to apply indirection operators in a variety of ways to maintain the correct **[in]** and **[out]** semantics. Typical declarations for a context handle are as follows. In the **.idl** file, declare a named type such as

```
typedef [context_handle] void* my_handle_t;
```

A manager routine that returns a context handle as an **out** parameter declares it as

```
my_handle_t *h;
```

The routine then sets the value of the handle as follows:

```
*h = &context_data;
```

A routine that refers to a context handle as an **in** parameter declares it as

```
my_handle_t h;
```

and dereferences the handle as follows:

```
context_data = (my_handle_t*)h;
```

For the **in,out** case, the routine uses the same declaration as in the **out** case, and dereferences the handle as follows:

```
context_data = (my_handle_t*)*h;
```

The following extensive example shows a simple use of context handles. In the sample code, the client requests a unit of storage from the server, using the **store_open()** call, and receives a handle to the allocated storage. The **store_read()**, **store_write()**, and **store_set_ptr()** routines allow the client to read from and write to specific locations in the allocated storage. The **store_close()** routine releases the server resources.

Context Handles in the Interface

The **.idl** file declarations for the **store** interface are as follows:

```
/*
 * store.idl
 * A sample interface that demonstrates server maintained context.
 * The client requests temporary storage of a specified size,
 * and the server returns a handle that can be used to read and
 * write to storage. The interface doesn't care how the
 * server implements the storage.
 */
[
uuid(0019b8c5-e8b5-1c84-9a41-0000c0d4de56),
pointer_default(ref),
version(1.0)
]
interface store
{

  /* A context handle used to access remote storage:        */
  typedef [context_handle] void* store_handle_t;

  /* A storage object name string:                 /
  /* typedef [string] char* store_name_t; */

  /* A buffer type for data:                    */
  typedef byte store_buf_t[*];

  /* Note that the context handle is an [out] parameter of the open */
  /* routine, an [in, out] parameter of the close routine, and an  */
  /* [in] parameter of the other routines. If the context handle  */
  /* were treated as an [in] parameter of the close routine, the  */
  /* stubs would never learn that the context had been set to NULL, */
  /* and would consider the context to still be live. This would  */
  /* result in the rundown routine's being called when the client  */
  /* terminated, even though there would be no context to run down. */

  void store_open(
[in] handle_t binding,
[in] unsigned32 store_size,
[out] store_handle_t *store_h,
[out] error_status_t *status
  );

  void store_close(
[in,out] store_handle_t *store_h,
[out] error_status_t *status
  );

  void store_set_ptr(
[in] store_handle_t store_h,
[in] unsigned32 offset,
[out] error_status_t *status
  );

  void store_read(
[in] store_handle_t store_h,
[in] unsigned32 buf_size,
[out, size_is(buf_size), length_is(*data_size)] \
 store_buf_t buffer,
[out] unsigned32 *data_size,
[out] error_status_t *status
  );

  void store_write(
[in] store_handle_t store_h,
[in] unsigned32 buf_size,
[in, size_is(buf_size)] store_buf_t buffer,
[out] unsigned32 *data_size,
[out] error_status_t *status
  );

}
```

Context Handles in a Server Manager

Server manager code to provide a rudimentary implementation of the **store** interface is as follows:

```
 /* context_manager.c -- implementation of "store" interface.   */
/*                                     */
/* The server maintains a certain number of storage areas, only one of */
/* which can be (or should be) opened by a single client at a time.  */
/* More than one client can, however, apparently be invoked (up to the */
/* number of separate storelets == store handles available, defined by */
/* the value of NUM_STORELETS). Each client keeps track of its store */
/* (and likewise enables the server to do the same) by means of the  */
/* context handle it receives when it opens its store.        */
/*                                     */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <pthread.h>
#include <dce/dce_error.h>
#include <dce/daclif.h>
```

```
#include "context.h"

#define NUM_STORELETS 10

/***********************************************************************/
/* The actual "storelet" structure...                  */

typedef struct store_hdr{
  pthread_mutex_t ref_lock;
  unsigned32 size;
  unsigned32 refcount;
  idl_byte *storage;
} store_hdr_t;

store_hdr_t headers[NUM_STORELETS]; /* There's an array of these.    */

/***********************************************************************/
/* The store specification structure; note that it is equivalent to the */
/* handle; the pointer to it is returned as the handle by the     */
/* store_open() routine below...                     */
/* The assumption is that all access to a given handle is serialized  */
/* in a single thread, so no locking is needed for these.       */

typedef struct store_spec{
  unsigned32 number;   /* The storelet number we've opened.     */
  unsigned32 offset;   /* The current read/write position.     */
} store_spec_t; /* There's only one of these; it's the handle that   */
         /* gives access to one of the NUM_STORELETS set of  */
         /* "storelets".                     */


/* The server entry name:                         */
extern unsigned_char_p_t entry;


/* Initialization control block:                     */
pthread_once_t init_once_blk = pthread_once_init;
/******
*
* store_mgmt_init -- Zeroes out all the storelet structures; executed
*          only once per server instance, as soon as a client
*          has called the store_open() routine.
*
******/
/***********************************************************************/
void
store_mgmt_init(
)
{
  int i;
  store_hdr_t *hdr;

  fprintf(stdout, "Store Manager: Initializing Store);
  memset(headers, 0, sizeof(store_hdr_t) * NUM_STORELETS);
  for (i = 0; i < NUM_STORELETS; i++)
  {
hdr = headers + i;
pthread_mutex_init(
  (pthread_mutex_t *)hdr,
  pthread_mutexattr_default);
  }

}

/******
*
* store_open -- Opens a store and returns a handle to it. Store consists
*        of one "storelet" selected from array of NUM_STORELETS.
*
******/
/*************************************************************************/
void
store_open(
  handle_t binding,
  unsigned32 store_size,  /* Size specified for actual storage.   */
  store_handle_t *store_h, /* To return the store handle in.     */
  error_status_t *status
)
{
  int i;         /* Index variable.              */
  store_spec_t *spec;  /* Store specification == handle.      */
  store_hdr_t *hdr;   /* Storelet structure.            */

  /* Do the store initialization if this is the first open call...  */
  /* Zero out the store headers...                  */
  pthread_once(&init_once_blk, store_mgmt_init);
  /* The following loop goes through all the storelets, looking for  */
  /* one whose reference count is zero. As soon as one such is   */
  /* found, a handle is allocated for it, storage is allocated for */
  /* its store structure, and the loop (and the call) terminates. If */
  /* no unreferenced storelet is found, a status of -1 is returned  */
  /* and no handle is allocated...                  */
  for(i = 0; i < NUM_STORELETS; i++)
  {
/* Go to the next storelet...                */
hdr = headers + i;

/* Is it unreferenced?...                  */
if (hdr->refcount == 0)
{
  /* If so, lock the header...            */
  *status = pthread_mutex_lock((pthread_mutex_t *)hdr);
  if (*status != 0)
  {
return;
  }

  /* ...and check the reference count again...    */
  if (hdr->refcount == 0)
  {
/* Now we know we "really" have this one.  */
/* Only one open is allowed, so lock only  */
/* the reference count...         */
hdr->refcount++;

/* Now unlock the header so other threads  */
/* can continue to check it...       */
*status = pthread_mutex_unlock((pthread_mutex_t *)hdr);
if (*status != 0)
  return;

/* Now allocate space for the specifica-  */
/* tion structure...            */
spec = (store_spec_t *)malloc(sizeof(store_spec_t));
spec->number = i;
spec->offset = 0;
*store_h = spec;

/* Allocate space for the storage part of  */
/* the header...             */
hdr->storage = (idl_byte *)malloc(store_size);
hdr->size = store_size;
   /* Finally, set the return status to OK,  */
   /* and return...              */
*status = error_status_ok;
return;
  }

  /* If the reference count turned out to have    */
  /* been accessed between our first check and our  */
  /* locking the mutex, we must now unlock the mutex */
  /* preparatory to looping around to check the next */
  /* storelet...                   */
  *status = pthread_mutex_unlock((pthread_mutex_t *)hdr);
  if (*status != 0)
  {
return;
  }
}
  }

  /* The following is reached only if we never found a free      */
  /* storelet...                           */
  *store_h = NULL;
  *status = -1;

}

/******
*
* store_set_ptr -- Insert a new value into the store buffer pointer.
*
*******/
/*********************************************************************/
void store_set_ptr(
  store_handle_t store_h,   /* The store handle.          */
  unsigned32 offset,  /* Value to insert into store buffer pointer. */
  error_status_t *status
)
{
  store_spec_t *spec;       /* Our pointer to store handle.  */

  spec = (store_spec_t *)store_h; /* Get the store spec.       */
  spec->offset = offset;   /* Copy in the new buffer pointer value. */
  *status = error_status_ok;
}
/******
*
* store_close -- Close the opened storelet.
*
******/
/*********************************************************************/
void
store_close(
  store_handle_t *store_h,        /* Store handle.      */
  error_status_t *status
)
{
  store_spec_t *spec;       /* Our pointer to store handle.   */
  store_hdr_t *hdr;        /* Pointer to a storelet.      */

  printf("Store Manager: Closing Store);

  spec = (store_spec_t *)*store_h; /* Get the store spec.       */
  hdr = headers + spec->number;  /* Point to the correct storelet. */

  /* If the thing is actually opened, close it...           */
  if (hdr->refcount > 0)
  {
/* Lock the header first...                 */
*status = pthread_mutex_lock((pthread_mutex_t *)hdr);
```

```
if (*status != 0)
{
  printf("Close: lock failed);
  return;
}

/* Check the reference count to make sure no one slipped in */
/* before we could lock the header, and already closed the */
/* critter...                       */
if (hdr->refcount > 0)
{
  /* The store is open, and it's locked by us, so we */
  /* can safely close it. So do it. First, decrement */
  /* the reference count...            */
  hdr->refcount--;

  /* Is it completely closed now?           */
  if (hdr->refcount == 0)
  {
/* If so, get rid of its storage space...  */
hdr->size = 0;
free(hdr->storage);
  }
}
/* If the store turned out to be closed before we could   */
/* close it, we have nothing to do but release the lock... */
*status = pthread_mutex_unlock((pthread_mutex_t *)hdr);
if (*status != 0)
{
  printf("Close: unlock failed);
  return;
}
  }

  /* And free our handle space...                   */
  free(spec);

  /* Be sure to NULL the context handle. Otherwise, the context   */
  /* will be considered to be live as long as the client is run-   */
  /* ning...                             */
  *store_h = NULL;
  *status = error_status_ok;
}

/******
*
* store_read -- Read a certain number of bytes from the opened store.
*
******/
/*************************************************************************/
void
store_read(
  store_handle_t store_h,  /* Store handle.             */
  unsigned32 buf_size,   /* Number of bytes to read.        */
  store_buf_t buffer,    /* Space to return data read in.     */
  unsigned32 *data_size,  /* To return number of bytes read in.   */
  error_status_t *status
)
{
  store_spec_t *spec;    /* Our handle pointer.          */
  store_hdr_t *hdr;     /* Pointer to a storelet.         */

  spec = (store_spec_t *)store_h; /* Get the storelet spec.     */
  hdr = headers + spec->number;  /* Point to the correct storelet. */

  /* If the amount we're to read is less than the amount left to be  */
  /* read, then read it...                      */
  if (buf_size <= hdr->size)
  {

/* Copy bytes from the storelet storage, beginning at off- */
/* set, into the return buffer, up to the size of the   */
/* buffer...                         */
memcpy(buffer, hdr->storage + spec->offset, buf_size);

/* Update the storelet buffer pointer past what we've just */
/* read...                          */
spec->offset += buf_size;

/* Show return size of data read...             */
*data_size = buf_size;
*status = error_status_ok;
return;
  }

  /* If there's less data left than has been specified to read, don't */
  /* read it...                             */
  *data_size = 0;
  *status = -1;
}

/******
*
* store_write -- Write some data into the opened store.
*
******/

void
store_write(
  /* handle_t IDL_handle,*/ /* If the server ACF declares      */
               /* [explicit_handle]          */
  store_handle_t store_h,  /* Store handle.             */
  unsigned32 buf_size,    /* Number of bytes to write.       */
  store_buf_t buffer,    /* Data to be written.          */
  unsigned32 *data_size,   /* To return number of bytes written.  */
  error_status_t *status
)
{
  store_spec_t *spec;      /* Our pointer to store handle.   */
  store_hdr_t *hdr;       /* Pointer to a storelet.      */

  /* Do an access check on IDL_handle here...             */
  /* [--ORIGINAL NOTE] -- I don't know what the above means.     */

  spec = (store_spec_t *)store_h; /* Get the storelet spec.     */
  hdr = headers + spec->number;  /* Point to the correct storelet. */

  /* If the amount of unused room left in the storelet is greater   */
  /* than what we're supposed to write in it, write it...      */
  if ((hdr->size - spec->offset) > buf_size)
  {

/* Copy bytes from the buffer into the storelet storage,  */
/* beginning at the current read/write position...     */
memcpy(hdr->storage + spec->offset, buffer, buf_size);

/* Update the storelet buffer pointer to point past what  */
/* we've just written...                  */
spec->offset += buf_size;

/* Add a null in case we want to read the store as a    */
/* string...                         */
*(hdr->storage + spec->offset) = 0;
/* Show return size of data written...           */
*data_size = buf_size;
*status = error_status_ok;
return;
  }

  /* If we don't have room to write the whole buffer, don't write   */
  /* anything...                             */
  *data_size = 0;
  *status = error_status_ok;
}

/******
 *
 * print_manager_error-- Manager version. Prints text associated with
 * bad status code.
 *
 *
 ******/
void
print_manager_error(
char *caller, /* String identifying routine that received the error. */
error_status_t status) /* status we want to print the message for.  */
{
  dce_error_string_t error_string;
  int print_status;

  dce_error_inq_text(status, error_string, &print_status);
  fprintf(stderr," Manager: %s: %s, caller, error_string);

}
```

The sample implementation of the store interface is obviously too limited for any practical use, but it does demonstrate the application of context handles in a straightforward way. A context handle returned by the **store_open()** routine is opaque to the client. To the server, it is a pointer to the server's representation of a storage unit. In this case, it points to a structure that keeps track of the client's current location within a specific piece of server-maintained storage.

Aside from deallocating the actual storage, the **store_close()** routine sets the context handle to NULL. The NULL value indicates to the server stub that the context is no longer active, and the stub, in turn, tells the RPC runtime not to maintain the context. For example, after the **store_close()** routine has been invoked, the rundown routine will not be invoked if communication ends between client and server. The context rundown routine takes care of closing the client's storage in case of a communication failure while the context is active.

The global array of *store_hdr* structures that keeps track of allocated storage, obviously servers no practical purpose in the example. (Presumably the operating system is already doing this!) However, it does provide a demonstration of the fact that global server manager data is shared data in the implicitly multithreaded server environment. The routines that manipulate this shared data may be called simultaneously by multiple server threads (in response to multiple simultaneous client calls); therefore, locking must be provided, in this case on the *refcount* field. The sample also demonstrates how the **pthread_once()** facility can be used to provide one-time initialization of the shared data on the first **store_open()** call.

As an exercise, the storage interface can easily be made more interesting by providing multiple clients simultaneous access to a given storage area. To implement this, the application could add a *store_name* parameter to the **store_open()** routine and replace the *refcount* field with counts of readers and writers. The division of the storage management between the *store_hdr* and the *store_spec* data structures is intended to facilitate this; the *store_hdr* holds shared state relating to each store, while the *store_spec* holds each thread's private state.

Context Rundown

Context handles typically point to some state maintained by a server instance for a client over a series of RPC operations. If the series of operations fails to complete because communication is lost between client and server, the server will probably have to take some kind of recovery action such as restoring data to a consistent state and freeing resources.

The stub detects outstanding context when it marshals context handle parameters. Outstanding context is considered to exist from the point at which a non-NULL pointer value is returned, until a NULL pointer value is returned. When outstanding context exists, the server stub code will call a context rundown routine in response to certain exceptions that indicate a loss of contact with the client. You should note that the exact timing of the call depends on the transport. In particular, with the connectionless protocol, servers that maintain context for clients expect clients to indicate periodically that they are still running. If the server fails to hear from the client during a specified timeout period, the server will assume that the client has stopped and call the context rundown routine. This can mean a substantial delay between the time the client actually fails and the time at which context maintained for the client is actually cleaned up. If the context being held represents a scarce resource on the server, one consequence of the delayed rundown may be that failed calls continue to hold the scarce resource for some time before it is made available again.

Since a context handle may be freely shared among threads of the calling client context, it is possible for outstanding context to exist for more than one call simultaneously. Such shared context is considered to be outstanding as long as it is outstanding for any of the participating threads. Also, any communications failures are likely to be detected at different times for each such call thread, and the difference in timing may be especially noticeable in the case of the connectionless protocol. Context rundown occurs only after all server call threads have been terminated. This means that call operations in progress on the server need not be concerned that the context they are operating on will be changed unexpectedly. Imagine a situation in which context handles represent open file descriptors, and the rundown routine closes the files. A manager thread that shares these descriptors via a context handle is guaranteed that the files will remain open even if a communications failure is detected in another thread that also is using the same context handle.

```
/******
*
* store_handle_t_rundown -- Closes the opened storelet.
*
******/
/**********************************************************/
void
store_handle_t_rundown(
  store_handle_t store_h
)
{
  error_status_t st;

  printf("Store Manager: Running down context.");
  store_close(&store_h, &st);
}
```

Binding and Security Information

One element that is clearly missing from the context handle sample code is any access checking. To do this, it is necessary to get the client binding, although it may not be immediately obvious how to do this with a context handle. The answer is actually quite simple but, to understand it, it helps to have a clear idea of how binding parameters operate in RPC.

Every call requires binding information, whether this is supplied explicitly as a binding parameter or not. When a call is made with a binding handle, the client uses cached binding information associated with the binding handle. When no binding handle parameter is passed, the client derives the binding information it needs by some other means. For example, with a context handle, the client uses cached binding information associated with the context handle.

Even when an explicit binding handle parameter is present, the handle is not marshalled as call data in the same way other call parameters are. Similarly, on the server side, when a binding handle parameter is present in a manager operation, it is unmarshalled simply as a reference to the binding information cached by the server runtime for the call. It is irrelevant whether the call was made with an explicit binding handle parameter on the client side.

Therefore, it is perfectly possible for a server manager operation to have a binding handle as a parameter even when the client RPC call is made without an explicit binding parameter.

The mechanics of this are to use different **.acf** declarations on the client and server sides. The **.idl** file declaration for the operation does not declare an explicit binding handle parameter, but the server **.acf** file applies the **[explicit_handle]** attribute to the operation. This results in a server stub that expects to unmarshal a binding handle as the first parameter of the operation, while the client stub does not expect an explicit binding handle parameter for the call.

An example of a server-side **.acf** file for the store interface is as follows:

```
/* store.acf - server side
 * Unmarshal a client binding handle on each call
 */

interface store
{
  store_open();
  [explicit_handle]store_close();
  [explicit_handle]store_set_ptr();
  [explicit_handle]store_read();
  [explicit_handle]store_write();
}
```

You could achieve the same effect by using different **.idl** files for the client and server, but this is not recommended. The **.idl** file serves as the canonical representation of an interface and hence should be the same for all clients and servers.

This technique can be used in a number of ways; for example, to permit the client to use implicit binding while the server manager operations extract authorization information from a client binding handle. In the case of a context handle, the principle is the same. You use the server **.acf** declarations to add a binding parameter to the call on the server side. The client continues to call using the context handle, while the server manager receives the client binding as a first extra parameter.

In the case of the sample code, the client calls to the store interface remain the same, but the server manager implementations now contain an extra parameter. For example:

```
void
store_write(
  handle_t IDL_handle,
  store_handle_t store_h,
  unsigned32 buf_size,
  store_buf_t buffer,
  unsigned32 *data_size,
  error_status_t *status
)
{
  store_spec_t *spec;
  store_hdr_t *hdr;

  if (check_access(IDL_handle, sec_acl_perm_write) == 0)
  {
    *status = str_s_no_perms;
    return;
  }
  .
  .
  .
}
```

Pipes

Pipes are a mechanism for efficiently handling large quantities of data by overlapping the transfer and processing of data. Input data is transferred in chunks to the server for processing, and output data is processed by the server in chunks and transferred to the client. A pipe is declared in a type definition of an interface definition, and the data type is used as parameters in the operations of the interface. The server manager calls stub pipe support routines in a loop, and the client stub calls pipe support routines that the client application must provide.

One of the pipe support routines that the client must provide is an **alloc** routine, which allocates a buffer for each chunk of pipe data. Given that pipes are intended to process data asynchronously, consuming it as it arrives, the **alloc** routine should not just blindly allocate a new buffer each time it is called, since the net effect would be to allocate space for the whole stream. A reasonable approach is either to declare a buffer statically or allocate it on the first call (per thread), and thereafter simply return the same buffer. The following code example shows the form an **alloc** routine takes in client application code.

```
#define CLIENT_BUFFER_SIZE 2048
idl_byte client_buffer[CLIENT_BUFFER_SIZE];

void client_alloc (state, bsize, buf, bcount)
  rpc_ss_pipe_state_t state;
  unsigned int bsize;
  byte **buf;
  unsigned int *bcount;
```

```
{
  *buf = client_buffer;
  *bcount = CLIENT_BUFFER_SIZE;
}
```

Input Pipes

In the following example, a client sends the contents of a file to a server as a set of chunks allocated from the same static buffer. The chunks are processed (in this case simply printed) as they arrive.

The declaration in the interface definition is as follows:

```
typedef pipe char test_pipe_t;

void pipe_test1(
  [in] handle_t handle,
  [in] test_pipe_t test_pipe,
  [out] error_status_t *status
);
```

Note that the pipe is declared as a **typedef**, resulting in an IDL-generated C typedef for **test_pipe_t**, which is a structure containing pointers to the pipe support routines and a pipe state field. The server manager and client code then implement the pipe in a complementary fashion.

For an **[in]** pipe, the server manager code consists of a cycle of calls to the **test_pipe.pull** routine (a server stub routine) which terminates when a zero-length chunk is received:

```
void
pipe_test1(
  handle_t binding_h,
  test_pipe_t test_pipe,
  error_status_t *status
)
{
  char buffer[SBUFFSIZE];
  int count;
  char *cptr;
  do
  {
    (*(test_pipe.pull))(test_pipe.state, buffer, \
 SBUFFSIZE, &count);
    for (cptr = buffer; cptr < buffer + count; cptr++)
      putchar(*cptr);
  } while (count > 0);
}
```

Using the buffer supplied by the manager, the **test_pipe.pull** routine unmarshals an amount of data that is nonzero, but not more than the buffer can hold. There is no guarantee that the buffer will be filled. The actual amount of data in the buffer is indicated by the **count** parameter returned in the **test_pipe.pull** routine. This count equals the number of **test_pipe_t** data elements in the buffer.

The **test_pipe.pull** routine signals the end of data in the pipe by returning a chunk whose count is 0 (zero). Any attempt to pull data from the pipe after the zero-length chunk has been encountered will cause an exception to be raised. The **in** pipes must be processed in the order in which they occur in the operation signature. Attempting to pull data from an **in** pipe before end-of-data on any preceding **in** pipe has been encountered will result in an exception being raised. If the manager code attempts to write to an **out** pipe or return control to the server stub before end-of-data has been encountered on the last **in** pipe, an exception will be raised. (Note that there is no guarantee that chunks seen by the manager will match the chunks supplied by the client's **pull** routine.)

The client application code must supply **pull** and **alloc** routines and a pipe state. These routines must work together to produce a sequence of pointers to chunks, of which only the last is empty. In the following example, the client code provides a **test_pipe.pull** routine that reads chunks of the input file into a buffer and returns a count of the chunk size, returning a zero count when the end of the file is reached. The pipe state block is used here simply as a convenient way to make the file state available to the **pull** routine. Applications need not make any use of the pipe state.

```
/* Client declares types and routines */

typedef struct client_pipe_state_t {
  idl_char *filename;
  idl_boolean file_open;
  int file_handle;
} client_pipe_state_t;

client_pipe_state_t client_in_pipe_state = {false, 0};

void client_pull(state,buf,esize,ecount)
  client_pipe_state_t * state;
  byte *buf;
  unsigned int esize;
  unsigned int *ecount;
{
  if (! state->file_open)
  {
    state->file_handle = open(state->filename,O_RDONLY);
    if (state->file_handle == -1)
    {
      printf("Client couldn't open %s", state->filename);
      exit(0);
    }
    state->file_open = true;
  }
  *ecount = read(state->file_handle, buf, esize);
  if (*ecount == 0)
  {
    close(state->file_handle);
    state->file_open = false;
  }
}
```

Finally, the client must do the following:

1. Allocate the **test_pipe_t** structure.
2. Initialize the **test_pipe_t.pull**, **test_pipe_t.alloc**, and **test_pipe_t.state** fields.
3. Include code where appropriate for checking the **pipe_t.state** field.
4. Pass the structure as the pipe parameter. The structure can be passed either by value or by reference, as indicated by the signature of the operation that contains the pipe parameter:

```
/* Client initializes pipe */
test_pipe_t test_pipe;

test_pipe.pull = client_pull;
test_pipe.alloc = client_alloc;
test_pipe.state = (rpc_ss_pipe_state_t)&client_in_pipe_state;

/* Client makes call */

pipe_test1(binding_h, test_pipe, &status);
```

To transmit a large amount of data that is already in the proper form in memory (that is, the data is already an array of **test_pipe_t**), the client application code can have the **alloc** routine allocate a buffer that already has the information in it. In this case, the **pull** routine becomes a null routine.

Output Pipes

An **[out]** pipe is implemented in a similar way to an input pipe, except that the client and server make use of the **push** routine instead of the **pull** routine. The following samples show an **[out]** pipe used to read the output from a shell command executed by the server.

The declarations in the interface definition are as follows:

```
typedef pipe char test_pipe_t;

void pipe_test2(
  [in] handle_t handle,
  [in, string] char cmd[],
  [out] test_pipe_t *test_pipe,
  [out] error_status_t *status
);
```

The server manager routines demonstrate a couple of possible implementations. In each case, the manager makes a cycle of calls to the server stub's **push** routine, ending by pushing a zero-length chunk:

```
#include <dirent.h>
#define SBUFFSIZE 256

void
pipe_test2(
  handle_t binding_h,
  idl_char *cmd,
  test_pipe_t *test_pipe,
  error_status_t *status
)
{

  DIR *dir_ptr;
  struct dirent *directory;

  char buffer[SBUFFSIZE];
  FILE *str_ptr;
  int n;

  /* An elementary mechanism to execute a command and get
   * the output back. Note that popen() and fread() are
   * thread-safe, so the whole process won't block while
   * the call thread waits for them to return.
   *
   * This is potentially a dangerous operation!
   * Here we'll only allow a couple of "safe" commands.
   */

  if (!strcmp(cmd, "ps") || !strcmp(cmd, "ls"))
  {
    if ((str_ptr = popen(cmd, "r")) == NULL)
      return;
    while ((n = fread(buffer, sizeof(char), \
 SBUFFSIZE, str_ptr)) > 0)
    {
      (*(test_pipe->push))(test_pipe->state, buffer, n);
    }
    (*(test_pipe->push))(test_pipe->state, buffer, 0);
    fclose(str_ptr);
  }

  /* Here's another method: list an arbitrary directory
   * This time, we buffer the directory names as null-
   * terminated strings of various lengths. The client
   * will need to provide formatting of the output stream,
   * for example, by substituting a CR for each NULL byte.
   */

  /*
  if ((dir_ptr = opendir(cmd)) == NULL)
  {
    printf("Can't open directory %s, cmd);
    return;
  }
  while ((directory = readdir(dir_ptr)) != NULL)
  {
    if (directory->d_ino == 0)
      continue;
    (*(test_pipe->push))(test_pipe->state, \
 directory->d_name,
             strlen(directory->d_name)+1);
  }
  (*(test_pipe->push))(test_pipe->state, \
   directory->d_name, 0);
  closedir(dir_ptr);
  */

  *status = error_status_ok;
}
```

The stub enforces well-behaved pipe filling by the manager by raising exceptions as necessary. After all **in** pipes have been drained completely, the **out** pipes must be completely filled, in order.

The client code uses the same declarations as in the input pipe example, except that instead of using a **client_pull** routine it uses a **test_push** routine that prints out the contents of each received buffer:

```
/*
 * Our push routine prints each received buffer-full.
 */
void test_push(
  rpc_ss_pipe_state_t *state,
  idl_char *buf,
  unsigned32 count
)
{
  unsigned_char_t *cptr;
  for (cptr = buf; cptr < buf + count; cptr++)
  {
    /* For the second, directory reading example,
      uncomment the following:
    if (*cptr == 0)
      *cptr = ';
     */
    putchar(*cptr);
  }
}
```

For an **out** pipe, the client code must do the following:

1. Allocate the **test_pipe_t** structure.
2. Initialize the **test_pipe_t.push** and **test_pipe_t.state** fields.
3. Pass the structure as the pipe parameter, either by value or by reference.

```
test_pipe_t test_pipe;

test_pipe.alloc = (void (*)())client_alloc;
test_pipe.push = (void (*)())test_push;
test_pipe.state = (rpc_ss_pipe_state_t)&out_test_pipe_state;

pipe_test2(binding_h, cmd, &test_pipe, &status);
```

The client stub unmarshals chunks of the pipe into a buffer and calls back to the application, passing a reference to the buffer. To allow the application code to manage its memory usage, and possibly avoid unnecessary copying, the client stub first calls back to the application's **test_pipe.alloc** routine to get a buffer. In some cases, this may result in the **test_pipe.push** routine's not having any work to do.

The client stub may go through more than one (**test_pipe.alloc**, **test_pipe.push**) cycle in order to unmarshal data that the server marshalled as a single chunk. Note that there is no guarantee that chunks seen by the client stub will match the chunks supplied by the server's **push** routine.

Pipe Summary

The pipe examples show how the client and server tasks are complementary. The client implements the appropriate callback routines (**test_pipe.alloc** and either **test_pipe.push** or **test_pipe.pull**), and the server manager makes a cycle of calls to either **test_pipe.push** or **test_pipe.pull** of the stub. The application code gives the illusion that the server manager is calling the client-supplied callbacks. In fact, the manager is actually calling stub-supplied callbacks, and the client callbacks are asynchronous: a server manager call to one of the callback routines does not necessarily result in a call to the corresponding client callback.

One result of this is that the client and server should not count on the chunk sizes being the same at each end. For example, in the last directory reading example, the manager calls the **test_pipe.push** routine once with each NULL-terminated filename. However, the client **test_push** routine does not necessarily receive the data stream one filename at at time. For example, if the **test_push** routine attempted to print the filenames using **printf(″%s\n″,buf);**, it might fail. An interesting exercise would be to add **printf()** routines to the client callbacks and the server manager to show when each callback is made.

Note also that the use of the pipe *state* field by the client is purely local and entirely at the discretion of the client. The state is not marshalled between client and server, and the server stubs use the local *state* field in a private manner. The server manager should not alter the state field.

Pipes may also be **[in,out]**, although the utility of this construct is somewhat limited. Ideally, a client would like to be able to pass a stream of data to the server and have it processed and returned asynchronously. In practice, the input and output streams must be processed synchronously; that is, all input processing must be finished before any output processing can be done. This means that **[in, out]** pipes, while they can reduce latency within both the server and the client, cannot reduce latency between server and client; the client must still wait for all server processing to finish before it can begin to process the returned data stream.

For an **in,out** pipe, both the **pull** routine (for the **in** direction) and a **push** routine (for the **out** direction) must be initialized, as well as the **alloc** routine and the state. During the last **pull** call (when it will return a zero count to indicate that the pipe is drained), the application's **pull** routine must reinitialize the pipe state so that the pipe can be used by the **push** routine correctly.

Nested Calls and Callbacks

A called remote procedure can call another remote procedure. The call to the second remote procedure is nested within the first call; that is, the second call is a nested remote procedure call. A nested call involves the following general phases, as illustrated in Figure 51:

1. A client makes an initial remote procedure call to the first remote procedure.
2. The first remote procedure makes a nested call to the second remote procedure.
3. The second remote procedure executes the nested call and returns it to the first remote procedure.
4. The first remote procedure then resumes executing the initial call.

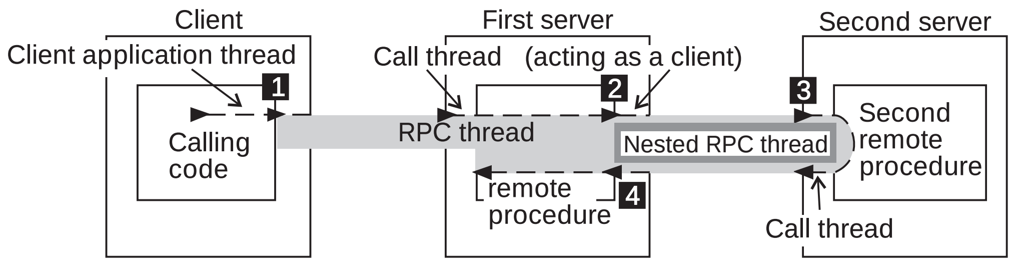


Figure 51. Phases of a Nested RPC Call

A specialized form of a nested remote procedure call involves a called remote procedure that is making a remote procedure call (callback) to the address space of the calling client application thread. Calling the client's address space requires that a server application thread be listening in that address space. Also, the second remote procedure needs a server binding handle for the address space of the calling client.

The remote procedure can ask the local RPC runtime to convert the client binding handle, provided by the server runtime, into a server binding handle. This is done by calling the **rpc_binding_server_from_client()** routine. This routine returns a partially bound binding handle (the server binding information lacks an endpoint). For a nested remote procedure call to find the address space of the calling client, the application must ensure that the partially bound binding handle is filled in with the endpoint of that address space. The the **rpc_binding_server_from_client(3rpc)** reference page discusses alternatives for ensuring that the endpoint is obtainable for a nested remote procedure call.

Using the server binding handle, a remote procedure can attempt a nested remote procedure call. The nested call involves the general phases illustrated by Figure 52.

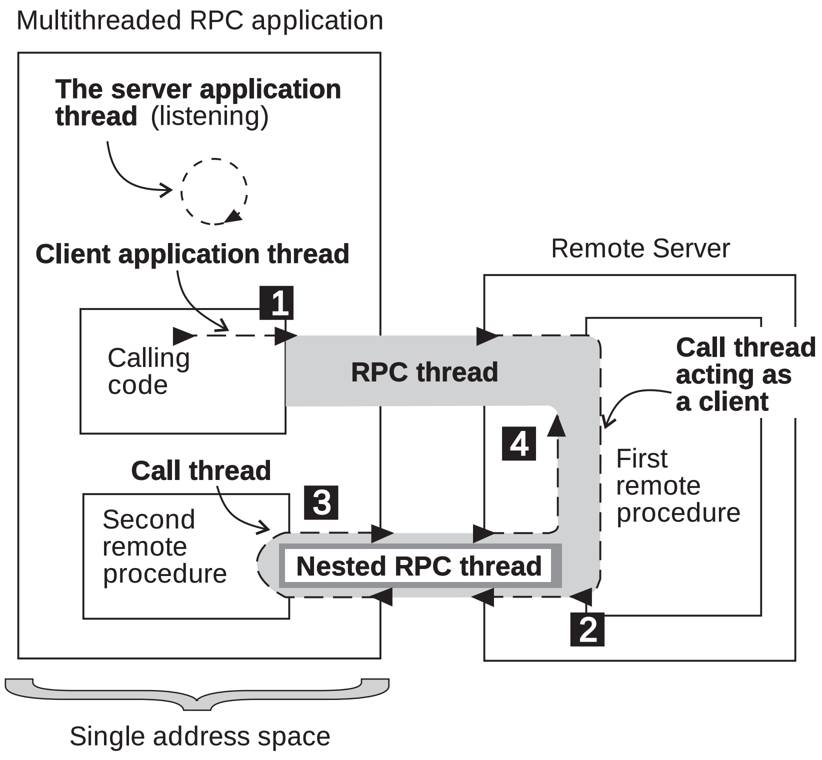


Figure 52. Phases of a Nested RPC Call to Client Address Space

The application threads in the preceding figure are performing the following activities:

1. A client application thread from a multithreaded RPC application makes an initial remote procedure call to the first remote procedure.
2. After converting the client binding handle into a server binding handle and obtaining the endpoint for the address space of the calling client application thread, the first remote procedure makes a nested call to the second remote procedure at that address space.
3. The second remote procedure executes the nested call and returns it to the first remote procedure.
4. The first remote procedure then resumes executing the initial call (the client).

Routing Remote Procedure Calls

The following section discusses routing incoming remote procedure calls between their arrival at a server's system and the server's invocation of the requested remote procedure. The following routing steps are discussed:

1. If a client has a partially bound server binding handle, before sending a call request to a server, the client runtime must get the endpoint of a compatible server from the endpoint mapper service of the server's system. This endpoint becomes the server address for a call request.
2. When the request arrives at the endpoint, the server's system places it in a request buffer belonging to the corresponding server.

3. As one of its scheduled tasks, the server gets the incoming calls from the request buffer. The server either accepts or rejects an incoming call, depending on available resources. If no call thread is available, an accepted call is queued to wait its turn for an available call thread.
4. The server then allocates an available call thread to the call.
5. The server identifies the appropriate manager for the called remote procedure and invokes the procedure in that manager to execute the call.
6. When the call thread finishes executing a call, the server returns the call's output arguments and control to the client.

Figure 53 illustrates these steps.

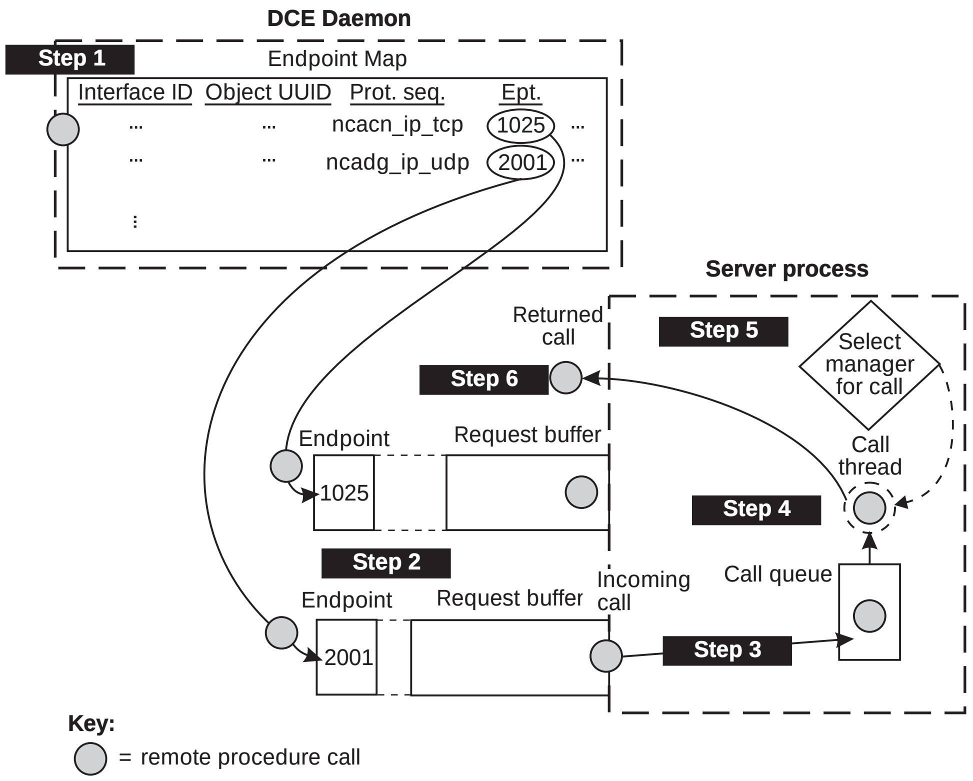


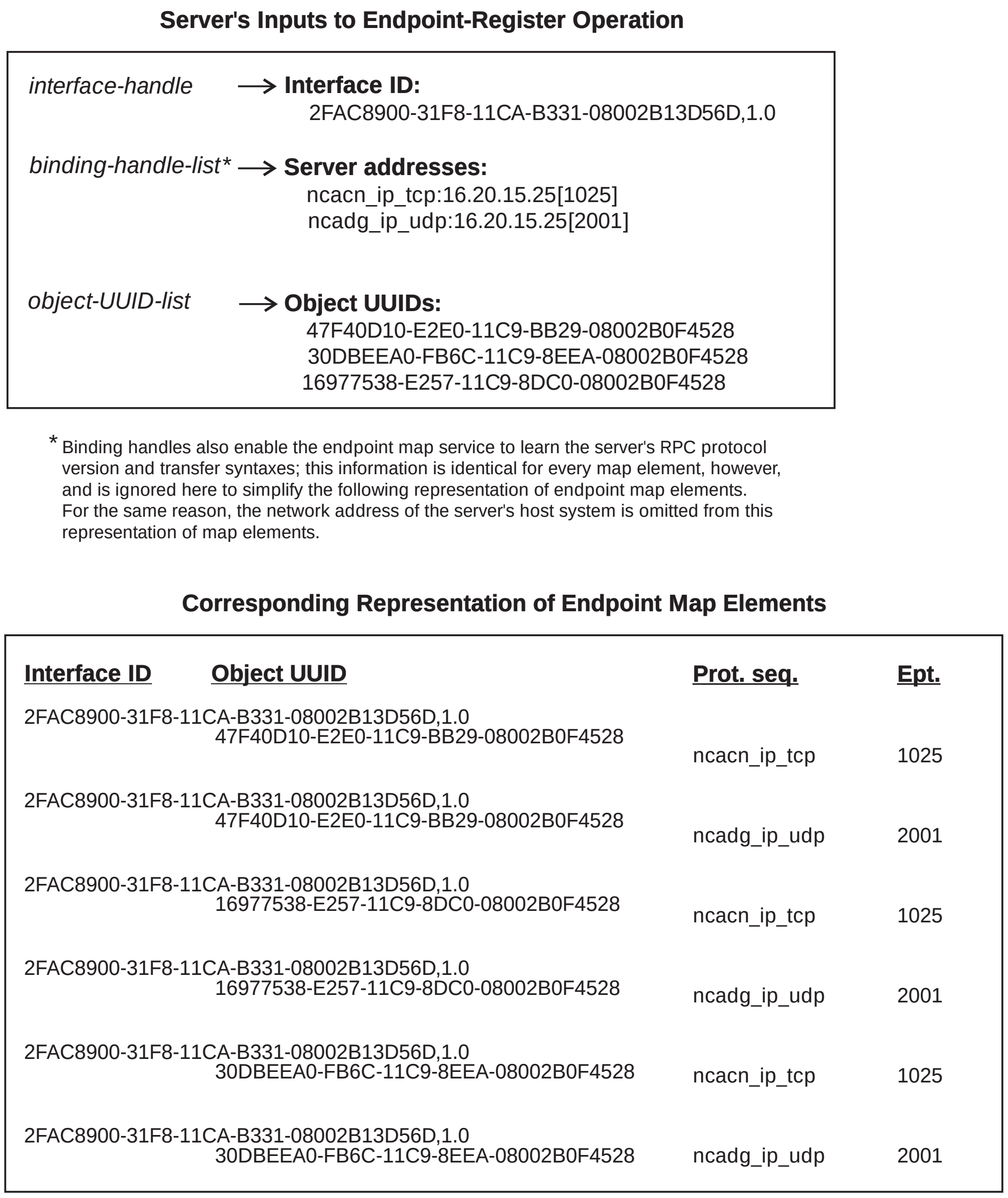
Figure 53. Steps in Routing Remote Procedure Calls

The concepts in the following subsections are for the advanced RPC developer. The first subsection discusses how clients obtain endpoints when using partially bound binding handles. Then we discuss how a system buffers call requests and how a server queues incoming calls; this information is relevant mainly to advanced RPC developers. The final subsection discusses how a server selects the manager to execute a call; it is relevant for developing an application that implements an interface for different types of RPC objects.

Obtaining an Endpoint

The endpoint mapper service of **dced** maintains the local endpoint map. The endpoint map is composed of elements which contain fully bound server binding information for a potential binding and an associated interface identifier and object UUID (which may be nil). Optionally, a map element can also contain an annotation such as the interface name.

Servers use the local endpoint mapper service to register their binding information. Each interface for which a server must register binding information requires a separate call to an **rpc_ep_register...()** routine, which calls the endpoint map service. The endpoint map service uses a new map element for every combination of binding information specified by the server. Figure 54 shows the correspondence between server binding information specified by a server and a graphic representation of the resulting endpoint map elements.



Server's Inputs to Endpoint-Register Operation

interface-handle $\longrightarrow$ **Interface ID:**
2FAC8900-31F8-11CA-B331-08002B13D56D,1.0

*binding-handle-list** $\longrightarrow$ **Server addresses:**
ncacn_ip_tcp:16.20.15.25[1025]
ncadg_ip_udp:16.20.15.25[2001]

object-UUID-list $\longrightarrow$ **Object UUIDs:**
47F40D10-E2E0-11C9-BB29-08002B0F4528
30DBEEA0-FB6C-11C9-8EEA-08002B0F4528
16977538-E257-11C9-8DC0-08002B0F4528

* Binding handles also enable the endpoint map service to learn the server's RPC protocol version and transfer syntaxes; this information is identical for every map element, however, and is ignored here to simplify the following representation of endpoint map elements. For the same reason, the network address of the server's host system is omitted from this representation of map elements.

Corresponding Representation of Endpoint Map Elements

Interface ID	Object UUID	Prot. seq.	Ept.
2FAC8900-31F8-11CA-B331-08002B13D56D,1.0	47F40D10-E2E0-11C9-BB29-08002B0F4528	ncacn_ip_tcp	1025
2FAC8900-31F8-11CA-B331-08002B13D56D,1.0	47F40D10-E2E0-11C9-BB29-08002B0F4528	ncadg_ip_udp	2001
2FAC8900-31F8-11CA-B331-08002B13D56D,1.0	16977538-E257-11C9-8DC0-08002B0F4528	ncacn_ip_tcp	1025
2FAC8900-31F8-11CA-B331-08002B13D56D,1.0	16977538-E257-11C9-8DC0-08002B0F4528	ncadg_ip_udp	2001
2FAC8900-31F8-11CA-B331-08002B13D56D,1.0	30DBEEA0-FB6C-11C9-8EEA-08002B0F4528	ncacn_ip_tcp	1025
2FAC8900-31F8-11CA-B331-08002B13D56D,1.0	30DBEEA0-FB6C-11C9-8EEA-08002B0F4528	ncadg_ip_udp	2001

Figure 54. Mapping Information and Corresponding Endpoint Map Elements

A remote procedure call made with server binding information that lacks an endpoint uses an endpoint from the endpoint map service. This endpoint must come from binding information of a compatible server. The map element of a compatible server contains the following:

- A compatible interface identifier

  The requested interface UUID and compatible version numbers are necessary. For the version to be compatible, the major version number requested by the client and registered by the server must be identical, and the requested minor version number must be less than or equal to the registered minor version number.

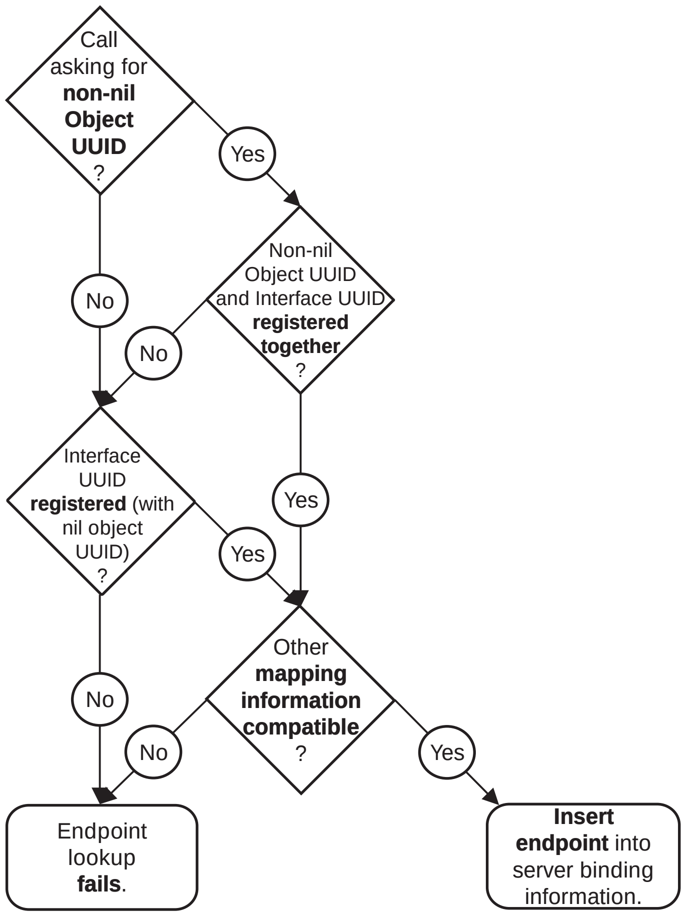
- The requested object UUID, if registered for the interface
- A server binding handle that refers to compatible binding information that contains the following:
  - A protocol sequence from the client's server binding information
  - The same RPC protocol major version number that the client runtime supports
  - At least one transfer syntax that matches one used by the client's system

To identify the endpoint of a compatible server, the endpoint service uses the following rules:

1. If the client requests a nonnil object UUID, the endpoint map service begins by looking for a map element that contains both the requested interface UUID and object UUID.
   a. On finding an element containing both of the UUIDs, the endpoint map service selects the endpoint from that element for the server binding information used by the client.
   b. If no element contains both UUIDs, the endpoint map service discards the object UUID and starts over (see rule 2).
2. If the client requests the nil object UUID (or if the requested nonnil object UUID is not registered), the endpoint map service looks for an element containing the requested interface UUID and the nil object UUID.
   a. On finding that element, the endpoint map service selects the endpoint from the element for the client's server binding information.
   b. If no such element exists, the lookup fails.

The RPC protocol service inserts the endpoint of the compatible server into the client's server binding information.

Figure 55 on page 343 illustrates the decisions the endpoint map service makes when looking up an endpoint for a client.



Call asking for **non-nil Object UUID** ?
Yes
No
Non-nil Object UUID and Interface UUID **registered together** ?
No
Yes
Interface UUID **registered** (with nil object UUID) ?
Yes
No
Other **mapping information compatible** ?
No
Yes
Endpoint lookup **fails**.
Insert endpoint into server binding information.

Figure 55. Decisions for Looking Up an Endpoint

You can design a server to allow the coexistence on a host system of multiple interchangeable instances of a server. Interchangeable server instances are identical, except for their endpoints; that is, they offer the same RPC interfaces and objects over the same network (host) address and protocol sequence pairs. For clients, identical server instances are fully interchangeable.

Usually, for each such combination of mapping information, the endpoint map service stores only one endpoint at a time. When a server registers a new endpoint for mapping information that is already registered, the endpoint map service replaces the old map element with the new one.

For interchangeable server instances to register their endpoints in the local endpoint map, they must instruct the endpoint map service not to replace any existing elements for the same interface identifier and object UUID. Each server instance can create new map elements for itself by calling the **rpc_ep_register_no_replace()** routine.

When a client uses a partially bound binding handle, load sharing among interchangeable server instances depends on the RPC protocol the client is using.

- Connectionless (datagram) protocol

  The map service selects the first map element with compatible server binding information. If necessary, a client can achieve a random selection among all the map elements with compatible binding information. However, this requires that, before making a remote procedure call, the client needs to resolve the binding by calling the **rpc_ep_resolve_binding()** routine.

- Connection-oriented protocol

  The client RPC runtime uses the **rpc_ep_resolve_binding()** routine, and the endpoint map service selects randomly among all the map elements of compatible servers.

For an alternative selection criteria, a client can call the **rpc_mgmt_ep_elt_inq_**{**begin**,**next**,**done**}**()** routines and use an application-specific routine to select from among the binding handles returned to the client.

When a server stops running, its map elements become outdated. Although the endpoint map service routinely removes any map element containing an outdated endpoint, a lag time exists when stale entries remain. If a remote procedure call uses an endpoint from an outdated map element, the call fails to find a server. To avoid clients getting stale data from the endpoint map, before a server stops, it should remove its own map elements.

A server also has the option of removing any of its own elements from the local endpoint map and continuing to run. In this case, an unregistered endpoint remains accessible to clients that know it.

Buffering Call Requests

Call requests for RPC servers come into the RPC runtime over the network. For each endpoint that a server registers (for a given protocol sequence), the runtime sets up a separate request buffer. A request buffer is a first-in, first-out queue where an RPC system temporarily stores call requests that arrive at an endpoint of an RPC server. The request buffers allow the runtime to continue to accept requests during heavy activity. However, a request buffer may fill up temporarily, causing the system to reject incoming requests until the server fetches the next request from the buffer. In this case, the calling client can try again, with the same server or a different server. The client does not know why the call is rejected, nor does the client know when a server is available again.

Each server process regularly dequeues requests, one by one, from all of its request buffers. At this point, the server process recognizes them as incoming calls. The interval for removing requests from the buffers depends on the activities of the system and of the server process.

How the runtime handles a given request depends partly on the communications protocol over which it arrives, as follows:

- A call over a connectionless transport is routed by the server’s system to the call request buffer for the endpoint specified in the call.
- A call over a connection-oriented transport may be routed by the server’s system to a request buffer or the call may go directly to the server process.

  Whether a remote procedure call goes to the request buffer depends on whether the client sends the call over an established connection. If a client makes a remote procedure call without an established connection, the server’s system treats the call request as a connection request and places the call request into a request buffer. If an established connection is available, the client uses it for the remote procedure call; the system handles the call as an incoming call and sends it directly to the server process that owns the connection.

Whether a server gets an incoming call from a request buffer or over an existing connection, the server process manages the call identically. A server process applies a clear set of call-routing criteria to decide whether to dispatch a call immediately, queue it, or reject it (if the server is extremely busy). These call-routing criteria are discussed in “Queuing Incoming Calls”.

When telling the RPC runtime to use a protocol sequence, a server specifies the number of calls it can buffer for the specified communications protocol (at a given endpoint). Usually, it is best for a server to specify a default buffer size, represented by a literal whose underlying value depends on the communications protocol. The default equals the capacity of a single socket used for the protocol by the server’s system.

The default usually is adequate to allow the RPC runtime to accept all the incoming call requests. For a well-known endpoint, the size of a request buffer cannot exceed the capacity of a single socket descriptor (the default size); specifying a higher number causes a runtime error. For well-known endpoints, specify the default for the maximum number of call requests.

For example, consider the request buffer at full capacity as represented in Figure 56. This buffer has the capacity to store five requests. In this example, the buffer is full, and the runtime rejects incoming requests, as is happening to the sixth request.

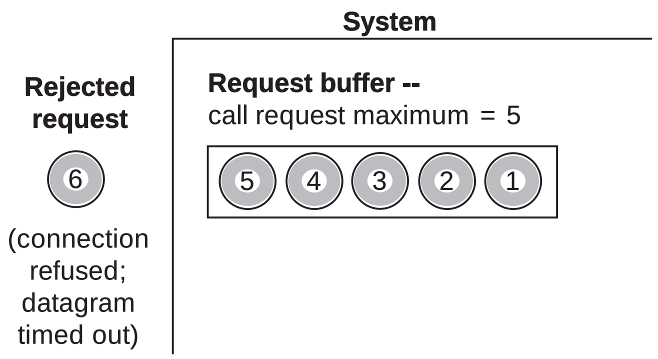


Figure 56. A Request Buffer at Full Capacity

Queuing Incoming Calls

Each server process uses a first-in, first-out call queue. When the server is already executing its maximum number of concurrent calls, it uses the queue to hold incoming calls. The capacity of queues for incoming calls is implementation dependent; most implementations offer a small queue capacity, which may be a multiple of the maximum number of concurrently executing calls.

A call is rejected if the call queue is full. The appearance of the rejected call depends on the RPC protocol the call is using, as follows:

- Connectionless (datagram) protocol

  The server does not notify the client about this failure. The call fails as if the server does not exist, returning an **rpc_s_comm_failure** communications status code (**rpc_x_comm_failure** exception).
- Connection-oriented protocol

The server rejects the call with an **rpc_s_server_too_busy** communications status code (**rpc_x_server_too_busy** exception).

The server process routes each incoming call as it arrives. Call routing is illustrated by the server in Figure 57. This server has the capacity to execute only one call concurrently. Its call queue has a capacity of eight calls. This figure consists of four stages (A through D) of call routing by a server process. On receiving any incoming call, the server begins by looking at the call queue.

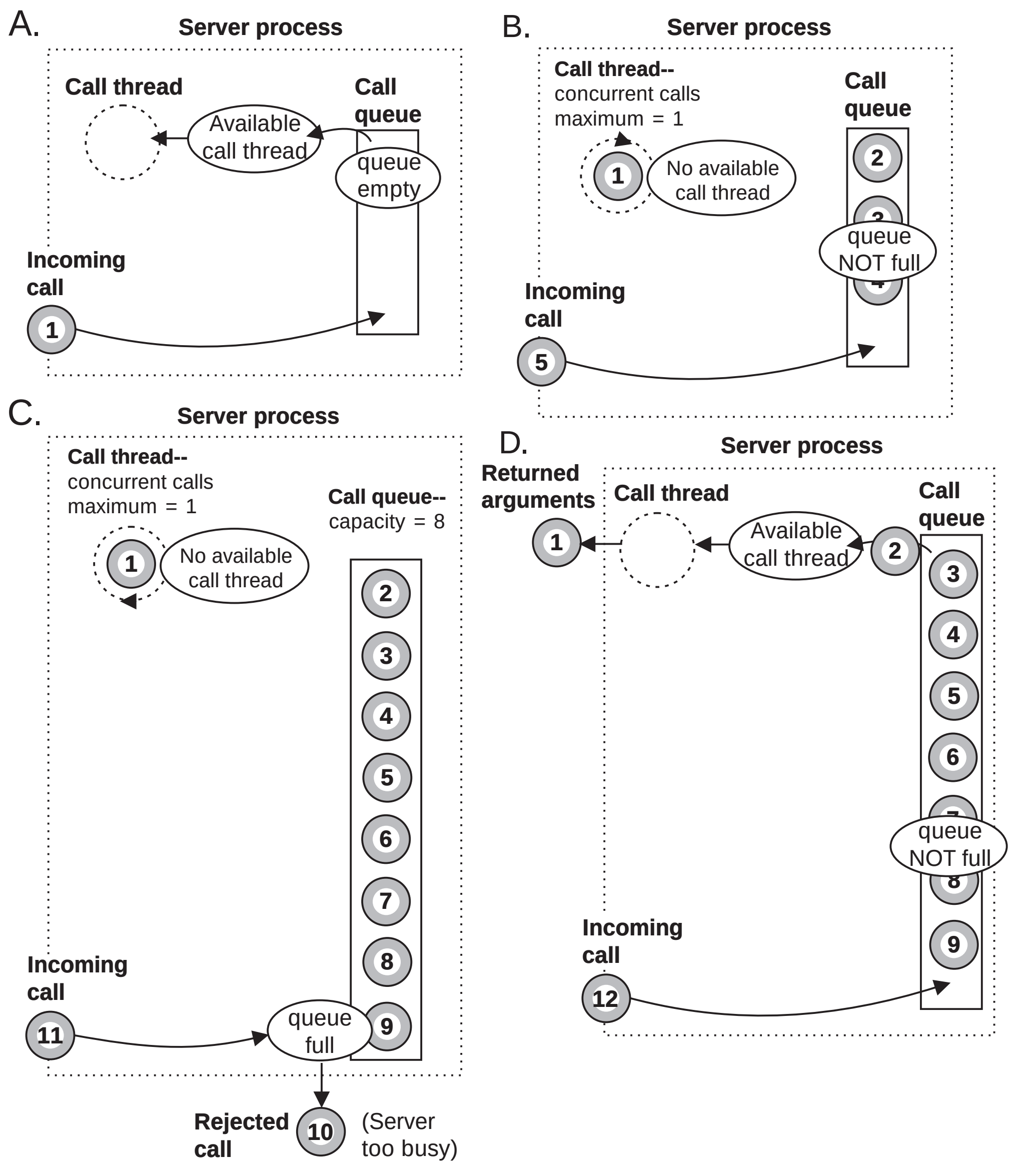


Figure 57. Stages of Call Routing by a Server Process

The activities of the four stages in the preceding figure are described as follows:

1. In stage **A**, call **1** arrives at a server that lacks any other calls. When the call arrives, the queue is empty and a call thread is available. The server accepts the call and immediately passes it to a call thread. The requested remote procedure executes the call in that thread, which becomes temporarily unavailable.

2. In stage **B**, call **5** arrives. The call queue is partially full, so the server accepts the call and adds it to the end of the queue.
3. In stage **C**, call **11** arrives. The queue is full, so the server rejects this call, as it rejected the previous call, **10**. (The caller can try again with the same or a different server.)
4. In stage **D**, the called procedure has completed call **1**, making the call thread available. The server has removed call **2** from the queue and is passing it to the call thread for execution. Thus, the queue is partially empty as call **12** arrives, so the server accepts the call and adds it to the queue.

Selecting a Manager

[2]Unless an RPC interface is implemented for more than one specific type of object, selecting a manager for an incoming call is a simple process. When registering an interface with a single manager, the server specifies the nil type UUID for the manager type. In the absence of any other manager, all calls, regardless of whether they request an object, go to the nil type manager.

The situation is more complex when a server registers multiple managers for an interface. The server runtime must select from among the managers for each incoming call to the interface. The DCE RPC dispatching mechanism requires a server to set a nonnil type UUID for a set of objects and for any interface that will access the objects in order to register a manager with the same type UUID.

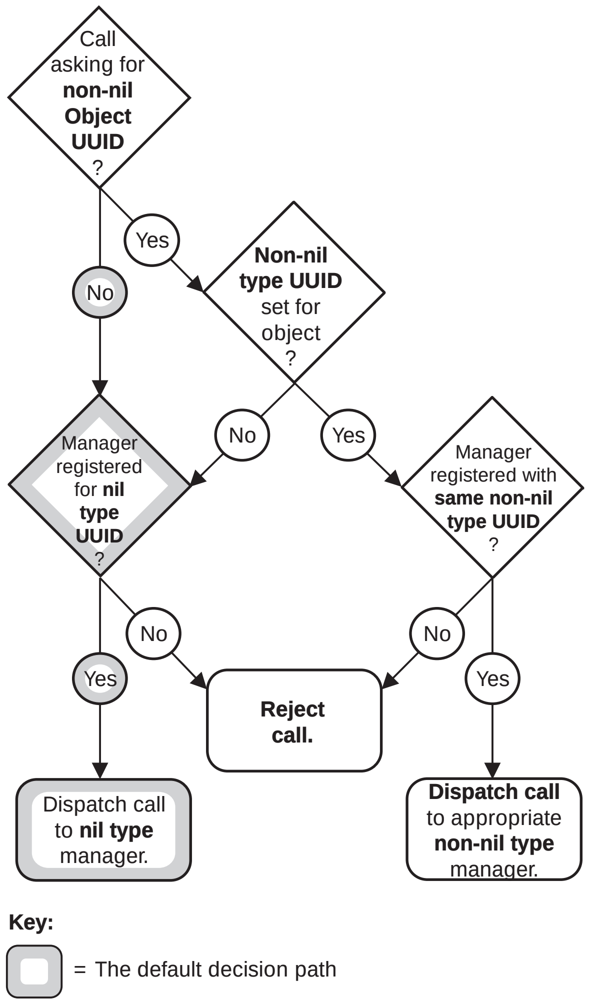
To dispatch an incoming call to a manager, a server does the following:

1. If the call contains the nil object UUID, the server looks for a manager registered with the nil type UUID (the nil type manager).
   a. If the nil type manager exists for the requested interface, the server dispatches the call to that manager.
   b. Otherwise, the server rejects the call.
2. If the call contains a nonnil object UUID, the server looks to see whether it has set a type for the object (by assigning a nonnil type UUID).

   If the object lacks a type, the server looks for the nil type manager.
   a. If the nil type manager exists for the requested interface, the server dispatches the call to that manager.
   b. Otherwise, the server rejects the call.
3. If the object has a type, the call requires a remote procedure of a manager whose type matches the object's type. In its absence, the RPC runtime rejects the call.

Figure 58 on page 348 illustrates the decisions a server makes to select a manager to which to dispatch an incoming call.

2. The API uses NULL to specify a synonym to the address of the nil UUID, which contains only zeros.



Call asking for **non-nil Object UUID** ?
Yes
No
Non-nil type UUID set for object ?
No
Yes
Manager registered for **nil type UUID** ?
Manager registered with **same non-nil type UUID** ?
No
No
Yes
Yes
Reject call.
Dispatch call to **nil type** manager.
Dispatch call to appropriate **non-nil type** manager.
Key:
= The default decision path

Figure 58. Decisions for Selecting a Manager

Creating Portable Data via the IDL Encoding Services

The IDL encoding services provide client and server RPC applications with a method for encoding data types in input parameters into byte stream format and decoding data types in output parameters from a byte stream without invoking the RPC runtime. Encoding and decoding functions are just like marshalling and unmarshalling, except that the data is stored locally and is not transmitted over the network; the IDL encoding services separate the data marshalling and unmarshalling functions from interaction with the RPC runtime.

Client and server applications can use the IDL encoding services to flatten (or serialize) a data structure, even binary data, and then store it; for example, by writing it to a file on disk. An RPC application on any DCE machine, regardless of its data type size and byte endianess, is then able to use the IDL encoding services to decode previously encoded data. Without the IDL encoding services, you cannot create a file of data on one machine and then successfully read that data on another machine that has different size data types and byte endianess.

The IDL encoding services can generate code that takes the input parameters to a procedure and places them in a standard form in one or more buffers that are delivered to user code. This process is called encoding. Encoded data can be written to a file or forwarded by a messaging system. The IDL encoding services can also generate code that delivers, as the output parameters of a procedure, data that has been converted into the standard form by encoding. Delivery of data in this way is called decoding. Data to be decoded can be read from a file or received by a messaging system.

Applications use the ACF attributes **encode** and **decode** as operation attributes or as interface attributes to direct the IDL compiler to generate IDL encoding services stubs for operations rather than generating RPC stubs. See “Chapter 19. Attribute Configuration Language” on page 425 for usage information on **encode** and **decode**.

Memory Management

IDL encoding services stubs handle memory management in the same way as RPC client stubs: when you call an operation to which the **encode** and/or **decode** attributes have been applied, the encoding services stub uses whatever client stub memory management scheme is currently in effect. See “Memory Management” on page 313 for further details on client stub memory management defaults and setting up memory management schemes.

You can control which memory management scheme the stubs will use by calling the **rpc_ss_swap_client_alloc_free()** and **rpc_ss_set_client_alloc_free()** routines. The first routine sets the memory management routines used by both the encoding and decoding stubs, and the second routine restores the previous memory management scheme after encoding and decoding are complete.

Note that the memory management scheme established, whether explicitly or by default, is on a per-thread basis.

Buffering Styles

There are a number of different ways in which buffers containing encoded data can be passed between the application code and the IDL encoding services. These are referred to as different buffering styles. The different buffering styles are:

- Incremental encoding

  The incremental encoding style requires that you provide an **allocate** routine which creates an empty buffer into which IDL encoding services can place encoded data, and a **write** routine which IDL encoding services will call when the buffer is full or all the parameters of the operation have been encoded. The IDL encoding services call the **allocate** and **write** routines repeatedly until the encoding of all of the parameters has been delivered to the user code. See the **idl_es_encode_incremental(3rpc)** reference page for a description of the required parameters for the **allocate** and **write** routines.
- Fixed buffer encoding

  The fixed buffer encoding style requires that the application supply a single buffer into which all the encoded data is to be placed. The buffer must have an address that is 8-byte aligned and must be a multiple of 8 bytes in size. It must also be large enough to hold an encoding of all the data, together with an encoding header for each operation whose parameters are being encoded; 56 bytes should be allowed for each encoding header.

- Dynamic buffer encoding

  With the dynamic buffer encoding style, the IDL encoding services build a single buffer containing all the encoded data and deliver the buffer to application code. The buffer is allocated by whatever client memory management mechanism has been put in place by the application code. The default for this is **malloc()**. When the application code no longer needs the buffer, it should release the memory resource.

  The dynamic buffer encoding style has performance implications. The IDL encoding services will usually allocate a number of intermediate buffers, then allocate the buffer to be delivered to the application code, copy data into it from the intermediate buffers, and release the intermediate buffers.
- Incremental decoding

  The incremental decoding buffering style requires that you provide a **read** routine which, when called, delivers to the IDL encoding services a buffer that contains the next part of the data to be decoded. The IDL encoding services will call the **read** routine repeatedly until all of the required data has been decoded. See the **idl_es_encode_incremental(3rpc)** reference page for a description of the required parameters for the **read** routine.
- Buffer decoding

  The buffer decoding style requires that you supply a single buffer containing all the encoded data. Where application performance is important, note that, if the supplied buffer is not 8-byte aligned, the IDL encoding services allocate a temporary aligned buffer of comparable size and copy data from the user-supplied buffer into it before performing the requested decoding.

IDL Encoding Services Handles

When an application's encoding or decoding operation is invoked, the handle passed to it must be an IDL encoding services handle (the **idl_es_handle**_t type). The IDL encoding services handle indicates whether encoding or decoding is required, and what style of buffering is to be used. The IDL encoding services provides a set of routines to enable the application code to obtain encoding and decoding handles to the IDL encoding services. The IDL encoding services handle-returning routine you call depends on the buffering style you have chosen:

- If you have selected the incremental encoding style, you call the **idl_es_encode_incremental()** routine, which returns an incremental encoding handle.
- If you have selected the fixed buffer encoding style, you call the **idl_es_encode_fixed_buffer()** routine, which returns a fixed buffer encoding handle.
- If you have selected dynamic buffer encoding, you call the **idl_es_encode_dyn_buffer()** routine, which returns a dynamic buffer encoding handle.
- If you have selected incremental decoding as your buffering style, you call the **idl_es_decode_incremental()** routine, which returns an incremental decoding handle.
- If you have selected the buffer decoding style, you call the **idl_es_decode_buffer()** routine, which returns a buffer decoding handle.

When the encoding or decoding for which an IDL encoding services handle was required is completed, the application code should release the handle resources by calling the **idl_es_handle_free()** routine. See the *OSF DCE Application Development Reference* for a complete description of the IDL encoding service routines.

It is an error to call an operation for which **encode** or **decode** has been specified by using an RPC binding handle, and it is an error to call an RPC operation by using an IDL encoding services handle.

The following restrictions apply to the use of IDL encoding services handles:

- An operation can be called with an encoding handle only if the operation has been given the **encode** ACF attribute.
- An operation can be called with a decoding handle only if the operation has been given the **decode** ACF attribute.
- The **auto_handle** ACF attribute cannot be used with the IDL encoding services.
- The **implicit_handle** ACF attribute cannot be used with the IDL encoding services.
- Customized handles cannot be used with the IDL encoding services.
- An **in** context handle does not contain the handle information needed by the IDL encoding services.

Programming Example

The following example uses the IDL encoding service features described in the preceding sections. The example verifies that the results of a number of decoding operations are the same as the parameters used to create the corresponding encodings.

The interface definition for this example is as follows:

```
[uuid(20aac780-5398-11c9-b996-08002b13d56d), version(0)]
interface es_array
{
  const long N = 5000;

  typedef struct
  {
    byte b;
    long l;
  } s_t;

  typedef struct
  {
    byte b;
    long a[7];
  } t_t;

  void in_array_op1([in] handle_t h, [in] long arr[N]);
  void out_array_op1([in] handle_t h, [out] long arr[N]);

  void array_op2([in] handle_t h, [in,out] s_t big[N]);

  void array_op3([in] handle_t h, [in,out] t_t big[N]);
}
```

The attribute configuration file for the example is as follows:

```
interface es_array
{
  [encode] in_array_op1();
  [decode] out_array_op1();
  [encode, decode] array_op2();
  [encode, decode] array_op3();
}
```

The test code for the example is as follows:

```
#include <dce/pthread_exc.h>
#include "rpcexc.h"
#include <stdio.h>
#include <stdlib.h>
#include <file.h>
#include <sys/file.h>
#include "es_array.h"

/*
 * User state for incremental encode/decode
 */
typedef struct es_state_t {
  idl_byte *malloced_addr;
  int file_handle;
} es_state_t;

static es_state_t es_state;

#define OUT_BUFF_SIZE 2048
static idl_byte out_buff[OUT_BUFF_SIZE];
static idl_byte *out_data_addr;
static idl_ulong_int out_data_size;

/*
 * User allocate routine for incremental encode
 */
void es_allocate(state, buf, size)
idl_void_p_t state;
idl_byte **buf;
idl_ulong_int *size;
{
  idl_byte *malloced_addr;
  es_state_t *p_es_state = (es_state_t *)state;

  malloced_addr = (idl_byte *)malloc(*size);
  p_es_state->malloced_addr = malloced_addr;
  *buf = (idl_byte *)(((malloced_addr - \
      (idl_byte *)0) + 7) & (~ 7));
  *size = (*size - (*buf - malloced_addr)) & (~ 7);
}
/*
 * User write routine for incremental encode
 */
void es_write(state, buf, size)
idl_void_p_t state;
idl_byte *buf;
idl_ulong_int size;
{
  es_state_t *p_es_state = (es_state_t *)state;

  write(p_es_state->file_handle, buf, size);
  free(p_es_state->malloced_addr);
}

/*
 * User read routine for incremental decode
 */
void es_read(state, buf, size)
idl_void_p_t state;
idl_byte **buf;
```

```
idl_ulong_int *size;
{
  es_state_t *p_es_state = (es_state_t *)state;

  read(p_es_state->file_handle, out_data_addr, out_data_size);
  *buf = out_data_addr;
  *size = out_data_size;
}

static ndr_long_int arr[N];
static ndr_long_int out_arr[N];
static s_t sarr[N];
static s_t ref_sarr[N];
static s_t out_sarr[N];
static t_t tarr[N];
static t_t ref_tarr[N];
static t_t out_tarr[N];
static ndr_long_int (*oarr)[M];

#define FIXED_BUFF_STORE (8*N+64)
static idl_byte fixed_buff_area[FIXED_BUFF_STORE];
/*
 * Test Program
 */
main()
{
  idl_es_handle_t es_h;
  idl_byte *fixed_buff_start;
  idl_ulong_int fixed_buff_size, encoding_size;
  idl_byte *dyn_buff_start;
  error_status_t status;
  int i,j;

  for (i = 0; i < N; i++)
  {
    arr[i] = random()%10000;
    sarr[i].b = i & 0x7f;
    sarr[i].l = random()%10000;
    ref_sarr[i] = sarr[i];
    tarr[i].b = i & 0x7f;
    for (j = 0; j < 7; j++) tarr[i].a[j] = random()%10000;
    ref_tarr[i] = tarr[i];
  }

  /*
   *Incremental encode/decode
   */
  /* Encode data using one operation */
  es_state.file_handle = open("es_array_1.dat", \
         O_CREAT|O_TRUNC|O_WRONLY, 0777);
  if (es_state.file_handle < 0)
  {
    printf("Can't open es_array_1.dat\n");
    exit(0);
  }
  idl_es_encode_incremental((idl_void_p_t)&es_state, es_allocate,
\
        es_write, &es_h, &status);
  if (status != error_status_ok)
  {
    printf("Error %08x from idl_es_encode_incremental\n", status);
    exit(0);
  }
  in_array_op1(es_h, arr);
  close(es_state.file_handle);
  idl_es_handle_free(&es_h, &status);
  if (status != error_status_ok)
```

```
{
  printf("Error %08x from idl_es_handle_free\n", status);
  exit(0);
}
/* Decode the data using another operation with          */
/* the same signature                       */
out_data_addr = (idl_byte *)(((out_buff - (idl_byte *)0) + 7) & (~7));
out_data_size = (OUT_BUFF_SIZE - (out_data_addr - out_buff)) & (~7);
es_state.file_handle = open("es_array_1.dat", O_RDONLY, 0);
if (es_state.file_handle < 0)
{
  printf("Can't open es_array_1.dat for reading\n");
  exit(0);
}
idl_es_decode_incremental((idl_void_p_t)&es_state, es_read,
              &es_h, &status);
if (status != error_status_ok)
{
  printf("Error %08x from idl_es_decode_incremental\n", status);
  exit(0);
}
out_array_op1(es_h, out_arr);
close(es_state.file_handle);
idl_es_handle_free(&es_h, &status);
if (status != error_status_ok)
{
  printf("Error %08x from idl_es_handle_free\n", status);
  exit(0);
}

/* Check the input and output are the same */
for (i = 0; i < N; i++)
{
  if (out_arr[i] != arr[i])
  {
    printf("out_arr[%d] - found %d - expecting %d\n",
        i, out_arr[i], arr[i]);
  }
}

/*
 * Fixed buffer encode/decode
 */
fixed_buff_start = (idl_byte *)(((fixed_buff_area - \
                       (idl_byte *)0) + 7)
                             &(~7));
fixed_buff_size = (FIXED_BUFF_STORE - \
              (fixed_buff_start - fixed_buff_area))
                             & (~7);
idl_es_encode_fixed_buffer(fixed_buff_start, fixed_buff_size,
              &encoding_size, &es_h, &status);
if (status != error_status_ok)
{
  printf("Error %08x from idl_es_encode_fixed_buffer\n", status);
  exit(0);
}
array_op2(es_h, sarr);
idl_es_handle_free(&es_h, &status);
if (status != error_status_ok)
{
  printf("Error %08x from idl_es_handle_free\n", status);
  exit(0);
}
idl_es_decode_buffer(fixed_buff_start, encoding_size, &es_h, &status);
if (status != error_status_ok)
{
  printf("Error %08x from idl_es_decode_buffer\n", status);
```

```
    exit(0);
  }
  array_op2(es_h, out_sarr);
  idl_es_handle_free(&es_h, &status);
  if (status != error_status_ok)
  {
    printf("Error %08x from idl_es_handle_free\n", status);
    exit(0);
  }
  for (i = 0; i < N; i++)
  {
    if (out_sarr[i].b != ref_sarr[i].b)
    {
      printf("array_op2 - out_sarr[%d].b = %c\n", i, out_sarr[i].b);
    }
    if (out_sarr[i].l != ref_sarr[i].l)
    {
      printf("array_op2 - out_sarr[%d].l = %d\n", i, out_sarr[i].l);
    }
  }

  /*
   * Dynamic buffer encode - fixed buffer decode
   */
  idl_es_encode_dyn_buffer(&dyn_buff_start, &encoding_size, &es_h, \
                              &status);
  if (status != error_status_ok)
  {
    printf("Error %08x from idl_es_encode_dyn_buffer\n", status);
    exit(0);
  }
  array_op3(es_h, tarr);
  idl_es_handle_free(&es_h, &status);
  if (status != error_status_ok)
  {
    printf("Error %08x from idl_es_handle_free\n", status);
    exit(0);
  }
  idl_es_decode_buffer(dyn_buff_start, encoding_size, &es_h, &status);
  if (status != error_status_ok)
  {
    printf("Error %08x from idl_es_decode_buffer\n", status);
    exit(0);
  }
  array_op3(es_h, out_tarr);
  rpc_ss_free (dyn_buff_start);
  idl_es_handle_free(&es_h, &status);
  if (status != error_status_ok)
  {
    printf("Error %08x from idl_es_handle_free\n", status);
    exit(0);
  }
  for (i = 0; i < N; i++)
  {
    if (out_tarr[i].b != ref_tarr[i].b)
    {
      printf("array_op3 - out_tarr[%d].b = %c\n", i, out_tarr[i].b);
    }
    for (j=0; j<7; j++)
    {
      if (out_tarr[i].a[j] != ref_tarr[i].a[j])
      {
        printf("array_op3 - out_tarr[%d].a[%d] = %d\n",
            i, j, out_tarr[i].a[j]);
      }
    }
  }

  printf("Test Complete\n");
}
```

Performing Multiple Operations on a Single Handle

Multiple operations can be performed using one encoding handle before the handle is released. In this case, all the encoded data is part of the same buffer system.

A single decoding handle is used to obtain the contents of the encoded data. Decoding operations must be called in the same order the encoding operations were called to create the encoded data.

The definition of the user client memory management functions, and any memory allocated by IDL encoding services using the client memory allocator, must not be modified between operations for which the same encoding handle is used.

Determining the Identity of an Encoding

Applications can use the **idl_es_inq_encoding_id()** routine to determine the identity of an encoding operation, for example, before calling their decoding operations.

Chapter 18. Interface Definition Language

This chapter describes how to construct an Interface Definition Language (IDL) file. First, it describes the IDL syntax notation conventions and lexical elements. It then describes the interface definition structure and the individual language elements supported by the IDL compiler.

The Interface Definition Language File

The IDL file defines all aspects of an interface that affect data passed over the network between a caller (client) and a callee (server). An interface definition file has the suffix **.idl**. In order for a caller and callee to interoperate, they both need to incorporate the same interface definition.

Syntax Notation Conventions

In addition to the documentation conventions described in the Preface of this guide, the IDL syntax uses the special notation described in the following subsections.

Typography

IDL documentation uses the following typefaces:

Bold **Bold** typeface indicates a literal item. Keywords and literal punctuation are represented in bold typeface. Identifiers used in a particular example are represented in bold typeface when mentioned in the text.

Italic *Italic* typeface indicates a symbolic item for which you need to substitute a particular value. In IDL syntax descriptions, all identifiers that are not keywords are represented in italic typeface.

Constant width
Constant width typeface is used for source code examples (in IDL or in C) that are displayed separately from regular text.

Special Symbols

IDL documentation uses the following symbolic notations:

[item] *Italic* brackets surrounding an item, which may include brackets in regular typeface, indicate that the item contained within them is optional.

[*item***]** Brackets shown in regular typeface surrounding a variable *item* indicate that the brackets are a required when the item is included, whether or not the item itself is required.

item **...**
Ellipsis points following an item indicate that the item may occur one or more times.

*item***, ...**
If an item is followed by a literal punctuation character and then by ellipsis points, the item may occur either once without the punctuation character or more than once with the punctuation character separating each instance.

... If ellipsis points are shown on a line by themselves, the item or set of items in the preceding line may occur any number of additional times.

item **|** *item*
If several items are shown separated by vertical bars, exactly one of those items must occur.

IDL Lexical Elements

The following subsections describe these IDL lexical elements:

- Identifiers
- Keywords
- Punctuation characters
- Whitespace
- Case sensitivity

Identifiers

The character set for IDL identifiers comprises the alphabetic characters A to Z and a to z, the digits 0 to 9, and the _ (underscore) character. An identifier must start with an alphabetic character.

No IDL identifier can exceed 31 characters. In some cases, an identifier has a shorter maximum length because the IDL compiler uses the identifier as a base from which to construct other identifiers; we identify such cases as they occur.

Keywords

IDL reserves some identifiers as keywords. In the text of this chapter, keywords are represented in **bold** typeface, and identifiers chosen by application developers are represented in *italic* typeface.

Punctuation Characters

IDL uses the following graphic characters:

```
" ' ( ) * , . / : ; | = [ \ ] { }
```

The **{** (left brace) and **}** (right brace) characters are national replacement set characters that may not be available on all keyboards. Wherever IDL specifies a left brace, the **??<** trigraph may be substituted. Wherever IDL specifies a right brace, the **??>** trigraph may be substituted.

Use of these trigraph sequences adds the following punctuation characters to the set in the preceding list:

```
< > ?
```

Whitespace

Whitespace is used to delimit other constructs. IDL defines the following whitespace constructs:

- A space

- A carriage return
- A horizontal tab
- A form feed at the beginning of a line
- A comment
- A sequence of one or more of the preceding whitespace constructs

A keyword, identifier, or number not preceded by a punctuation character must be preceded by whitespace. A keyword, identifier, or number not followed by a punctuation character must be followed by whitespace. Unless we note otherwise, any punctuation character may be preceded and/or followed by whitespace.

When enclosed in ″″ (double quotes) or **''** (single quotes), whitespace constructs are treated literally. Otherwise, they serve only to separate other lexical elements and are ignored.

Just as in C, the character sequence **/*** (slash and asterisk) begins a comment, and the character sequence ***/** (asterisk and slash) ends a comment. For example:

```
/* all natural */
import "potato.idl"; /* no preservatives */
```

Comments do not nest.

Case Sensitivity

The IDL compiler does not force the case of identifiers in the generated code.

The only case sensitivity issue that you have to be aware of is the implications involved in calling generated stubs from languages other than C.

IDL Versus C

IDL resembles a subset of ANSI C. The major difference between IDL and C is that there are no executable statements in IDL.

Declarations

An interface definition specifies how operations are called, not how they are implemented. IDL is therefore a purely declarative language.

Data Types

To support applications written in languages other than C, IDL defines some data types that do not exist in C and extends some data types that do exist in C. For example, IDL defines a Boolean data type.

Some C data types are supported by IDL only with modifications or restrictions. For example, unions must be discriminated, and all arrays must be accompanied by bounds information.

Attributes

The stub modules that are generated from an interface definition require more information about the interface than can be expressed in C. For example, stubs must know whether an operation parameter is an input or an output.

The additional information required to define a network interface is specified via IDL attributes. IDL attributes can apply to types, to structure members, to operations, to operation parameters, or to the interface as a whole. Some attributes are legal in only one of the preceding contexts; others are legal in more than one context. An attribute is always represented in **[ ]** (brackets) before the item to which it applies. For example, in an operation declaration, inputs of the operation are preceded by the **in** attribute and outputs are preceded by the **out** attribute:

```
void arith_add (
   [in] long a,
   [in] long b,
   [out] long *c,
   );
```

Interface Definition Structure

An interface definition has the following structure:

```
[interface_attribute, ...] interface interface_name
{

declarations
}
```

The portion of an interface definition that precedes the **{** (left brace) is the interface header. The remainder of the definition is the interface body. Interface header syntax and interface body syntax are described separately in the following two subsections.

Interface Definition Header

The interface header comprises a list of interface attributes enclosed in **[ ]** (brackets), the keyword **interface**, and the interface name:

```
[interface_attribute, ...] interface interface_name
```

Interface names, together with major and minor version numbers, are used by the IDL compiler to construct identifiers for interface specifiers, entry point vectors, and entry point vector types. If the major and minor version numbers are single digits, the interface name can be up to 17 characters long.

For C++ output, the interface header can also inherit an interface by using the inheritance operator (:) as follows:

```
[interface_attribute, ...] interface interface_name : inherited_interface
```

Interface Definition Body

The *declarations* in an interface definition body are one or more of the following:

```
import_declaration
constant_declaration
type_declaration
operation_declaration
```

A **;** (semicolon) terminates each declaration, and **{ }** (braces) enclose the entire body.

Import declarations must precede other declarations in the interface body. Import declarations specify the names of other IDL interfaces that define types and constants used by the importing interface.

Constant, type, and operation declarations specify the constants, types, and operations that the interface exports. These declarations can be coded in any order, provided any constant or type is defined before it is used.

Overview of IDL Attributes

Table 13 lists the attributes allowed in interface definition files and specifies the declarations in which they can occur.

Table 13. IDL Attributes

Attribute	**Where Used**
uuid	Interface definition headers
version	
endpoint	
exceptions	
pointer_default	
local	
broadcast	Operations
maybe	
idempotent	
reflect_deletions	
in	Parameters
out	
ignore	Structures
max_is	Arrays
min_is	
size_is	
first_is	
last_is	
length_is	

Table 13. IDL Attributes (continued)

Attribute	Where Used
string	Arrays
ptr	Pointers
ref	
unique	
handle	Customized handles
context_handle	Context handles
transmit_as	Type declarations

Interface Definition Header Attributes

The following subsections describe in detail the usage and semantics of the IDL attributes that can be used in interface definition headers. The attributes provided for interface definition headers are as follows:

- **uuid**
- **version**
- **endpoint**
- **exceptions**
- **pointer_default**
- **local**

The uuid Attribute

The **uuid** attribute specifies the Universal Unique Identifier (UUID) that is assigned to an interface. The **uuid** attribute takes the following form:

```
uuid (uuid_string)
```

A *uuid_string* is the string representation of a UUID. This string is typically generated as part of a skeletal interface definition by the utility **uuidgen**. A *uuid_string* contains one group of 8 hexadecimal digits, three groups of 4 hexadecimal digits, and one group of 12 hexadecimal digits, with hyphens separating the groups, as in the following example:

```
01234567-89ab-cdef-0123-456789abcdef
```

A new UUID should be generated for any new interface. If several versions of one interface exist, all versions should have the same interface UUID but different version numbers. A client and a server cannot communicate unless the interface imported by the client and the interface exported by the server have the same UUID. The client and server stubs in an application must be generated from the same interface definition or from interface definitions with identical **uuid** attributes.

Any remote interface must have the **uuid** attribute. An interface must have either the **uuid** attribute or the **local** attribute, but cannot have both.

The **uuid** attribute can appear at most once in an interface.

The following example illustrates use of the **uuid** attribute:

```
uuid(4ca7b4dc-d000-0d00-0218-cb0123ed9876)
```

The version Attribute

The **version** attribute specifies a particular version of a remote interface. The **version** attribute takes the following form:

version (*major [. minor]*)

A version number can be either a pair of integers (the major and minor version numbers) or a single integer (the major version number). If both major and minor version numbers are supplied, the integers should be separated by a period without whitespace. If no minor version number is supplied, 0 (zero) is assumed.

The following examples illustrate use of the **version** attribute:

```
version (1.1)  /* major and minor version numbers */

version (3)   /* major version number only */
```

The **version** attribute can be omitted altogether, in which case the interface is assigned 0.0 as the default version number.

A client and a server can communicate only if the following requirements are met:
- The interface imported by the client and the interface exported by the server have the same major version number.
- The interface imported by the client has a minor version number less than or equal to that of the interface exported by the server.

You must increase either the minor version number or the major version number when you make any compatible change to an interface definition. You must not decrease the minor version number unless you simultaneously increase the major version number.

You must increase the major version number when you make any incompatible change to an interface definition. (See the definition of compatible changes that follows.) You cannot decrease the major version number.

The following are considered compatible changes to an interface definition:
- Adding operations to the interface, if and only if the new operations are declared after all existing operation declarations in the interface definition.
- Adding type and constant declarations, if the new types and constants are used only by operations added at the same time or later. Existing operation declarations cannot have their signatures modified.

The *major* and *minor* integers in the **version** attribute can range from 0 to 65,535, inclusive. However, these typically are small integers and are increased in increments of one.

The following are considered incompatible changes to an interface definition:

- Changing the signature of an existing operation
- Changing the order of existing operations
- Adding a new operation other than at the end

The **version** attribute can appear at most once in an interface.

The endpoint Attribute

The **endpoint** attribute specifies the well-known endpoint or endpoints on which servers that export the interface will listen. The **endpoint** attribute takes the following form:

```
endpoint (endpoint_spec, ...)
```

Each *endpoint_spec* is a string in the following form:

```
" family : [endpoint] "
```

The *family* identifies a protocol family. The following are accepted values for *family*:

- **ncacn_ip_tcp**: NCA Connection over Internet Protocol: Transmission Control Protocol (TCP/IP)
- **ncadg_ip_udp**: NCA Datagram over Internet Protocol: User Datagram Protocol (UDP/IP)

The *endpoint* identifies a well-known endpoint for the specified *family*. The values accepted for *endpoint* depend on the *family* but typically are integers within a limited range. IDL does not define valid *endpoint* values.

Well-known endpoint values are typically assigned by the central authority that “owns” a protocol. For example, the Internet Assigned Numbers Authority assigns well-known endpoint values for the IP protocol family.

At compile time, the IDL compiler checks each *endpoint_spec* only for gross syntax. At runtime, stubs pass the *family* and *endpoint* strings to the RPC runtime, which validates and interprets them.

Most applications should not use well-known endpoints and should instead use dynamically assigned opaque endpoints. Most interfaces designed for use by applications should therefore not have the **endpoint** attribute.

The following example illustrates use of the **endpoint** attribute:

```
endpoint ("ncacn_ip_tcp:[1025]", "ncadg_ip_udp:[6677]")
```

The **endpoint** attribute can appear at most once in an interface.

The exceptions Attribute

The **exceptions** attribute specifies a set of user-defined exceptions that can be generated by the server implementation of the interface. The **exceptions** attribute takes the following form:

```
exceptions (exception_name [,exception_name] ...)
```

The following is a sample declaration of an **exceptions** attribute:

```
[uuid(06255501-08AF-11CB-8C4F-08002B13D56D),
version (1.1),
 exceptions (
   exc_e_exquota,
   binop_e_aborted,
   binop_e_too_busy,
   binop_e_shutdown)
] interface binop
 {
   long binop_add(
      [in] long a,
      [in] long b
      );
  }
```

See "Chapter 17. Topics in RPC Application Development" on page 313 for more information on using exceptions.

The pointer_default Attribute

IDL supports two kinds of pointer semantics. The **pointer_default** attribute specifies the default semantics for pointers that are declared in the interface definition. The **pointer_default** attribute takes the following form:

```
pointer_default (pointer_attribute)
```

Possible values for *pointer_attribute* are **ref, unique,** and **ptr**.

The default semantics established by the **pointer_default** attribute apply to the following usages of pointers:

- A pointer that occurs in the declaration of a member of a structure or a union.
- A pointer that does not occur at the top level of an operation parameter declared with more than one pointer operator. A top-level pointer is one that is not the target of another pointer and is not a field of a data structure that is the target of a pointer. (See "Pointer Attributes in Parameters" on page 394 for more information on top-level pointers.)

Note that the **pointer_default** attribute does not apply to a pointer that is the return value of an operation because this is always a full pointer.

The default semantics can be overridden by pointer attributes in the declaration of a particular pointer. If an interface definition does not specify **pointer_default** and contains a declaration that requires default pointer semantics, the IDL compiler will issue a warning. For additional information on pointer semantics, refer to "Unique Pointers" on page 393.

The **pointer_default** attribute can appear at most once in an interface.

The local Attribute

The **local** attribute indicates that an interface definition does not declare any remote operations and that the IDL compiler should therefore generate only header files, not stub files. The **local** attribute takes the following form:

local

An interface containing operation definitions must have either the **local** attribute or the **uuid** attribute. No interface can have both.

The **local** attribute can appear at most once in an interface.

Rules for Using Interface Definition Header Attributes

An interface cannot have both the **local** attribute and the **uuid** attribute. In an interface definition that contains any operation declarations, either **local** or **uuid** must be specified. In an interface definition that contains no operation declarations, both **local** and **uuid** can be omitted.

The **local**, **uuid**, and **version** attributes cannot be coded more than once. If the **endpoint** or the **pointer_default** attribute is coded more than once, the IDL compiler issues a warning and, where conflicts exist, the IDL compiler accepts the last value specified.

Examples of Interface Definition Header Attributes

The following example uses the **uuid** and **version** attributes:

```
[uuid(df961f80-2d24-11c9-be74-08002b0ecef1), version(1.1)]
interface my_interface_name
```

The following example uses the **uuid**, **endpoint**, and **version** attributes:

```
[uuid(0bb1a080-2d25-11c9-8d6e-08002b0ecef1),
endpoint("ncacn_ip_tcp:[1025]", "ncacn_ip_tcp:[6677]"), version(3.2)]
interface my_interface_name
```

Import Declarations

The IDL *import_declaration* specifies interface definition files that declare types and constants used by the importing interface. It takes the following form:

import *file*,... **;**

The *file* argument is the pathname, enclosed in double quotes, of the interface definition you are importing. This pathname can be relative; the **-I** option of the IDL compiler allows you to specify a directory from which to resolve import pathnames.

The effect of an import declaration is as if all constant, type, and import declarations from the imported file occurred in the importing file at the point where the import declaration occurs. Operation declarations are not imported.

For example, suppose that the interface definition **aioli.idl** contains a declaration to import the definitions for the **garlic** and **oil** interfaces:

```
import "garlic.idl", "oil.idl";
```

The IDL compiler will generate a C header file named **aioli.h** that contains the following **#include** directives:

```
#include "garlic.h"
#include "oil.h"
```

The stub files that the compiler generates will not contain code for any **garlic** and **oil** operations.

Importing an interface many times has the same effect as importing it once.

Constant Declarations

The IDL *constant_declaration* can take any one of the following forms:

```
const integer_type_spec identifier = integer | value | integer_const_expression;
const boolean identifier = TRUE | FALSE | value;
const char identifier = character | value;
const char* identifier = string | value;
const void* identifier = NULL | value;
```

The *integer_type_spec* is the data type of the integer constant you are declaring. The *identifier* is the name of the constant. The *integer*, *integer_const_expression*, *character*, *string*, or *value* specifies the value to be assigned to the constant. A *value* can be any previously defined constant.

IDL provides only integer, Boolean, character, string, and null pointer constants.

Following are examples of constant declarations:

```
const short TEN = 10;
const boolean FAUX = FALSE;
const char* DSCH = "Dmitri Shostakovich";
```

Integer Constants

An *integer_type_spec* is a *type_specifier* for an integer, except that the *int_size* for an integer constant cannot be **hyper**.

An*integer* is the decimal representation of an integer. IDL also supports the C notation for hexadecimal, octal, and long integer constants.

You can specify any previously defined integer constant as the *value* of an integer constant.

You can specify any arithmetic expression as the *integer_const_expression* that evaluates to an integer constant.

Boolean Constants

A Boolean constant can take one of two values: TRUE or FALSE.

You can specify any previously defined Boolean constant as the *value* of a Boolean constant.

Character Constants

A *character* is an ASCII character enclosed in single quotes. A white space character is interpreted literally. The **** (backslash) character introduces an escape sequence, as defined in the ANSI C standard. The **'** (single quote) character can be coded as the *character* only if it is escaped by a backslash.

You can specify any previously defined character constant as the *value* of a character constant.

String Constants

A *string* is a sequence of ASCII characters enclosed in double quotes. Whitespace characters are interpreted literally. The **** (backslash) character introduces an escape sequence, as defined in the ANSI C standard. The ″ (double quote) character can be coded in a *string* only if it is escaped by a backslash.

You can specify any previously defined string constant as the *value* of a string constant.

NULL Constants

A **void*** constant can take only one literal value: NULL.

You can specify any previously defined **void*** constant as the *value* of a **void*** constant.

Type Declarations

The IDL *type_declaration* enables you to associate a name with a data type and to specify attributes of the data type. It takes the following form:

```
typedef [[type_attribute, ...]] type_specifier type_declarator, ... ;
```

A *type_attribute* specifies characteristics of the type being declared.

The *type_specifier* can specify a base type, a constructed type, a predefined type, or a named type. A function pointer can be specified if the **local** attribute has been specified.

Each *type_declarator* is a name for the type being defined. Note, though, that a *type_declarator* can also be preceded by an ***** (asterisk), followed by **[]** (brackets), and can include **( )** (parentheses) to indicate the precedence of its components.

Type Attributes

A *type_attribute* can be any of the following:

- **handle**: The type being declared is a user-defined, customized-handle type.
- **context_handle**: The type being declared is a context-handle type.
- **transmit_as**: The type being declared is a presented type. When it is passed in remote procedure calls, it is converted to a specified transmitted type.
- **ref**: The type being declared is a reference pointer.
- **ptr**: The type being declared is a full pointer.

- **unique**: The type being declared is a unique pointer.
- **string**: The array type being declared is a string type.

Base Type Specifiers

IDL base types include integers, floating-point numbers, characters, a **boolean** type, a **byte** type, a **void** type, and a primitive handle type.

Table 14 lists the IDL base data type specifiers. Where applicable, the table shows the size of the corresponding transmittable type and the type macro emitted by the IDL compiler for resulting declarations.

Table 14. Base Data Type Specifiers

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

The base types are described individually later in this chapter.

Note that you can use the **idl_** macros in the code you write for an application to ensure that your type declarations are consistent with those in the stubs, even when the application is ported to another platform. The **idl_** macros are especially useful when passing constant values to RPC calls. For maximum portability, all constants passed to RPC calls declared in your network interfaces should be cast to the appropriate type because the size of integer constants (like the size of the **int** data type) is ambiguous in the C language.

The **idl_** macros are defined in **dce/idlbase.h**, which is included by header files that the IDL compiler generates.

Constructed Type Specifiers

IDL constructed types include structures, unions, enumerations, pipes, arrays, and pointers. (In IDL, as in C, arrays and pointers are specified via declarator constructs rather than type specifiers.) Following are the keywords used to declare constructed type specifiers:

```
struct
union
enum
pipe
```

Constructed types are described in detail later in this chapter.

Predefined Type Specifiers

While IDL per se does not have any predefined types, DCE RPC IDL implicitly imports **nbase.idl**, which does predefine some types. Specifically, **nbase.idl** predefines an error status type, several international character data types, and many other types. Following are the keywords used to declare these *predefined* type specifiers:

```
error_status_t
ISO_LATIN_1
ISO_MULTI_LINGUAL
ISO_UCS
```

The error status type and international characters are described in detail later in this chapter.

Type Declarator

An IDL *type_declarator* can be either a simple declarator or a complex declarator.

A simple declarator is just an identifier.

A complex declarator is an identifier that specifies an array, a function pointer, or a pointer.

Operation Declarations

The IDL *operation_declaration* can take the following forms:

```
[[operation_attribute, ...]] [static] type_specifier operation_identifier (parameter_declaration, .

[[operation_attribute, ...]] [static] type_specifier operation_identifier ([void]);
```

Use the first form for an operation that has one or more parameters; use the second form for an operation that has no parameters. Use the **static** keyword if the operation is a static member function of the interface class (C++ output only).

An *operation_attribute* can take the following forms:
- **idempotent**: The operation is idempotent.
- **broadcast**: The operation is always to be broadcast.
- **maybe**: The caller of the operation does not require and will not receive any response.
- **reflect_deletions**: If **rpc_ss_free()** is applied by application code on the server side to memory used for the referent of a full pointer that is part of an **[in]** parameter, the storage occupied by that referent on the client side is released.

- **ptr**: The operation returns a full pointer. This attribute must be supplied if the operation returns a pointer result and reference pointers are the default for the interface.
- **context_handle**: The operation returns a context handle.
- **string**: The operation returns a string.

The *type_specifier* in an operation declaration specifies the data type that the operation returns, if any. This type must be either a scalar type or a previously defined type. If the operation does not return a result, its *type_specifier* must be **void**.

For information on the semantics of pointers as operation return values, refer to the discussion of pointers in “Pointers” on page 390.

The *operation_identifier* in an operation declaration is an identifier that names the operation.

Each *parameter_declaration* in an operation declaration declares a parameter of the operation. A *parameter_declaration* takes the following form:

```
[parameter_attribute, ...] type_specifier parameter_declarator
```

Parameter declarations and the parameter attributes are described separately in the following sections.

Operation Attributes

Operation attributes determine the semantics to be applied by the RPC client and server protocol when an operation is called.

Operation Attributes: Execution Semantics

The **idempotent** attribute specifies that an operation is idempotent; that is, it can safely be executed more than once.

The **broadcast** attribute specifies that an operation is to be broadcast to all hosts on the local network each time the operation is called. The client receives output arguments from the first reply to return successfully, and all subsequent replies are discarded.

An operation with the **broadcast** attribute is implicitly idempotent.

Note that the broadcast capabilities of RPC runtime have a number of distinct limitations:

- Not all systems and networks support broadcasting. In particular, broadcasting is not supported by the RPC connection-oriented protocol.
- Broadcasts are limited to hosts on the local network.
- Broadcasts make inefficient use of network bandwidth and processor cycles.
- The RPC runtime library does not support **at-most-once** semantics for broadcast operations; it applies **idempotent** semantics to all such operations.
- The input arguments for broadcast calls are limited to 944 bytes.

The **maybe** attribute specifies that the caller of an operation does not expect any response. An operation with the **maybe** attribute cannot have any output parameters and cannot return anything. Delivery of the call is not guaranteed.

An operation with the **maybe** attribute is implicitly idempotent.

Operation Attributes: Memory Management

Use the **reflect_deletions** attribute to mirror the release of memory from server pointer targets to client pointer targets. When you use the **reflect_deletions** attribute, memory occupied by pointer targets on the client will be released when the corresponding pointer targets on the server are released. This is only true for pointer targets that are components of **[in]** parameters of the operation. By default, the mechanism used by RPC to release the pointer targets is the C language **free()** function unless the client code is executing as part of RPC server application code, in which case the **rpc_ss_free()** function is used. You can override the default by calling **rpc_ss_set_client_alloc_free()** or **rpc_ss_swap_client_alloc_free()** before the call to the remote operation.

Parameter Declarations

A *parameter_declaration* is used in an operation declaration to declare a parameter of the operation. A *parameter_declaration* takes the following form:

```
[parameter_attribute, ...] type_specifier parameter_declarator
```

If an interface does not use implicit handles or use interface-based binding, the first parameter must be an explicit handle that gives the object UUID and location. The handle parameter can be of a primitive handle type, **handle_t**, or a nonprimitive user-defined handle type.

A *parameter_attribute* can be any of the following:

- *array_attribute*: One of several attributes that specifies the characteristics of arrays.
- **in**: The parameter is an input attribute.
- **out**: The parameter is an output attribute.
- **ref**: The parameter is a reference pointer; it cannot be NULL and cannot be an aliased pointer.
- **ptr**: The parameter is a full pointer; it can be NULL and can be an aliased pointer.
- **unique**: The parameter is a unique pointer; it can be NULL.
- **string**: The parameter is a string.
- **context_handle**: The parameter is a context handle.
- **switch_is**:

The directional attributes **in** and **out** specify the directions in which a parameter is to be passed. The **in** attribute specifies that the parameter is passed from the caller to the callee. The **out** attribute specifies that the parameter is passed from the callee to the caller.

An output parameter must be passed by reference and therefore must be declared with an explicit ***** (asterisk). (Note that an array is implicitly passed by reference and so an output array does not require an explicit *****.) At least one directional attribute must be specified for each parameter of an operation.

An explicit handle parameter must have at least the **in** attribute.

The **ref, unique,** and **ptr** attributes are described later in “Pointers” on page 390. The **string** attribute is described in “Strings” on page 389. The **context_handle** attribute is described in “The context_handle Attribute” on page 404.

The *type_specifier* in a parameter declaration specifies the data type of the parameter.

The *declarator* in a parameter declaration can be any simple or complex declarator.

A parameter with the **out** attribute must be either an array or an explicitly declared pointer. An explicitly declared pointer is declared by a *pointer_declarator*, rather than by a *simple_declarator* with a named pointer type as its *type_specifier*.

For information on the semantics of pointers as operation parameters, refer to the discussion of pointers in “Pointers” on page 390.

Basic Data Types

The following subsections describe the basic data types provided by IDL and the treatment of international characters within IDL. The basic data types are as follows:

- Integer types
- Floating-point types
- The **char** type
- The **boolean** type
- The **byte** type
- The **void** type
- The **handle_t** type
- The **error_status_t**type

“Constructed Data Types” on page 376 describes the constructed data types that are built on the basic data types.

Integer Types

IDL provides four sizes of signed and unsigned integer data types, specified as follows:

```
int_size [int]
unsigned int_size [int]
int_size  unsigned [int]
```

The *int_size* can take the following values:

```
hyper

long
```

```
short

small
```

The **hyper** types are represented in 64 bits. A **long** is 32 bits. A **short** is 16 bits. A **small** is 8 bits.

The keyword **int** is optional and has no effect. The keyword **unsigned** denotes an unsigned integer type; it can occur either before or after the size keyword.

Floating-Point Types

IDL provides two sizes of floating-point data types, specified as follows:

```
float
double
```

A **float** is represented in 32 bits. A **double** is represented in 64 bits.

The char Type

The IDL character type is specified as follows:

```
[unsigned] char
```

A **char** is unsigned and is represented in 8 bits.

The keyword **unsigned** is optional and has no effect. IDL does not support a signed character type. IDL provides the **small** data type for representing signed 8-bit integers.

The boolean Type

The IDL **boolean** type is specified as follows:

```
boolean
```

A **boolean** is represented in 8 bits. A **boolean** is a logical quantity that assumes one of two values: TRUE or FALSE. Zero is FALSE and any nonzero value is TRUE.

The byte Type

The IDL **byte** type is specified as follows:

```
 byte
```

A **byte** is represented in 8 bits. The data representation format of **byte** data is guaranteed not to change when the data is transmitted by the RPC mechanism.

The IDL integer, character, and floating-point types (and hence any types constructed from these) are all subject to format conversion when they are transmitted between hosts that use different data representation formats. You can protect data of any type from format conversion by transmitting that type as an array of **byte**.

The void Type

The IDL **void** type is specified as follows:

```
void
```

The **void** type may be used to do the following:
- Specify the type of an operation that does not return a value
- Specify the type of a context handle parameter, which must be **void***
- Specify the type of a NULL pointer constant, which must be **void***

The handle_t Type

The IDL primitive handle type is specified as follows:

```
handle_t
```

A **handle_t** is a primitive handle type that is opaque to application programs but meaningful to the RPC runtime library. "Customized Handles" on page 403 discusses primitive and nonprimitive handle types.

The error_status_t Type

IDL provides the following predefined data type to hold RPC communications status information:

```
error_status_t
```

The values that can be contained in the **error_status_t** data type are compatible with the **unsigned long** and **unsigned32** IDL data types. These data types are used for status values in the DCE.

The **error_status_t** data type contains an additional semantic to indicate that this particular **unsigned long** contains a DCE format error status value. This additional semantic enables the IDL compiler to perform any necessary translation when moving the status value between systems with differing hardware architectures and software operating systems. If you are using status codes that are not in the DCE error status format or if you do not require such conversion, use an **unsigned long** instead of **error_status_t**.

International Characters

The implicitly imported **nbase.idl** provides predefined data types to support present and emerging international standards for the representation of characters and strings:

```
ISO_LATIN_1
ISO_MULTI_LINGUAL
ISO_UCS
```

Data of type **char** is subject to ASCII-EBCDIC conversion when transmitted by the RPC mechanism. The predefined international character types are constructed from the base type **byte** and are thereby protected from data representation format conversion.

The **ISO_LATIN_1** type is represented in 8 bits and is predefined as follows:

```
typedef byte ISO_LATIN_1;
```

The **ISO_MULTI_LINGUAL** type is represented in 16 bits and is predefined as follows:

```
typedef struct {
   byte row, column;
   } ISO_MULTI_LINGUAL;
```

The **ISO_UCS** type is represented in 32 bits and is predefined as follows:

```
typedef struct {
   byte group, plane, row, column;
   } ISO_UCS;
```

Constructed Data Types

The following subsections describe the constructed data types that are provided by IDL. The constructed types are built on the basic data types, which are described in "Basic Data Types" on page 373. The constructed data types are as follows:

- Structures
- Unions
- Enumerations
- Pipes
- Arrays
- Strings

In IDL, as in C, arrays and pointers are specified via declarator constructs. The other constructed types are specified via type specifiers.

Structures

The *type_specifier* for a structure type can take the following forms:

```
struct [tag] {

struct_member;
 ...
 }

struct tag
```

A *tag*, if supplied in a specifier of the first form, becomes a shorthand form for the set of member declarations that follows it. Such a *tag* can subsequently be used in a specifier of the second form.

A *struct_member* takes the following form:

```
[[struct_member_attribute, ...]] type_specifier declarator, ...;
```

A *struct_member_attribute* can be any of the following:

- *array_attribute*: One of several attributes that specify characteristics of arrays.

- **ignore**: An attribute indicating that the pointer member being declared is not to be transmitted in remote procedure calls.
- **ref**: An attribute indicating that the pointer member being declared is a reference pointer; it cannot be NULL and cannot be an alias.
- **ptr**: An attribute indicating that the pointer member being declared is a full pointer; it can be NULL and can be an alias.
- **unique**: An attribute indicating that the pointer member being declared is a unique pointer.
- **string**: An attribute indicating that the array member being declared is a string.
- **switch_is**:

A structure can contain a conformant array (conformant structure) only as its last member. And such a structure can be contained by another structure only as its last member, and so on. A conformant structure cannot be returned by an operation as its value and cannot be simply an **out** parameter. Note that a structure can contain any number of *pointer to* conformant arrays. Structure fields defined as pointers to an array base type and with one or more of the array size attributes define pointers to conformant arrays. Since the size of the pointer field in the structure is fixed, the structure itself is not conformant, although the array that it points to is conformant.

A structure cannot contain a pipe or context handle.

The **ignore** attribute specifies that the pointer is not to be transmitted in remote procedure calls. Note that the **ignore** attribute can be applied only to a pointer that is a member of a structure. The **ignore** attribute is not allowed in a type declaration that defines a pointer type.

Unions

IDL provides two types of unions: encapsulated and nonencapsulated. An IDL union must be discriminated. In an encapsulated union, the discriminator is part of the union. In a nonencapsulated union, the discriminator is not part of the union.

The following *type_specifier* can be used to indicate either kind of union.

```
union [tag]
```

A definition of the union identified by *tag* must appear elsewhere in the interface definition.

Encapsulated Unions

To define an encapsulated union, use the following syntax:

```
union [tag] switch (disc_type_spec discriminator) [union_name]
{

case ...

[default_case] }
```

If a *tag* is supplied, it can be used in a *type_specifier* of the form shown in “Unions”.

The *disc_type_spec* indicates the type of the *discriminator*, which can be an integer, a character, a **boolean**, or an enumeration.

The *union_name* specifies a name to be used in C code generated by the IDL compiler. When the IDL compiler generates C code to represent an IDL union, it embeds the union and its discriminator in a C structure. The name of the IDL union becomes the name of the C structure. If you supply a *union_name* in your type declaration, the compiler assigns this name to the embedded C union; otherwise, the compiler assigns the generic name **tagged_union**.

A *case* contains one or more labels and may contain a member declaration:

```
case constant:
...
 [union_member];
```

Each label in a *case* specifies a constant. The *constant* can take any of the forms accepted in an integer, character, or Boolean constant declaration, each of which is described earlier in this chapter.

A *default_case* can be coded anywhere in the list of cases:

```
default:

[union_member];
```

A *union_member* takes the following form:

```
[[union_member_attribute, ...]] type_specifier declarator;
```

A *union_member_attribute* can be any of the following:

- **ptr**: An attribute indicating that the pointer member being declared is a full pointer; it can be NULL and can be an alias.
- **string**: An attribute indicating that the character array member being declared is a string.

In any union, the type of the discriminator and the type of all constants in all case labels must resolve to the same type. At the time the union is used, the value of the discriminator selects a member, as follows:
- If the value of the discriminator matches the constant in any label, the member associated with the label is selected.
- If there is no label whose constant matches the value of the discriminator and there is a default case, the default member is selected.
- If there is no label whose constant matches the value of the discriminator and there is no default case, no member is selected and the exception **rpc_x_invalid_tag** is raised.

Note that IDL prohibits duplicate constant label values.

A *union_ member* can contain only one declarator. If no *union_member* is supplied, the member is NULL; if that member is selected when the union is used, no data is passed. But note that the discriminator is always passed.

A union cannot contain a pipe, a conformant array, a varying array, or any structure that contains a conformant or varying array. A union also cannot contain a **ref** or **unique** pointer or any structure that contains a **ref** or **unique** pointer.

The following is an example of an encapsulated union.

```
/* IDL construct /*

  typedef
    union fred switch (long a) ralph {
      case 1: float b;
      case 2: long c;
    } bill;

/* becomes in the generated header file /*

 typedef
   struct fred {
     long a;
     union {
       float b;
       long c;
     } ralph;
   } bill;
```

Nonencapsulated Unions

To define a nonencapsulated union, use the following syntax:

[switch_type(datatype)] **union** *[tag]*
{

case ...

[default_case] }

If a *tag* is supplied, it can be used in a *type_specifier* of the form shown in “Unions” on page 377.

A parameter or a structure field that is a nonencapsulated union must have an attribute attached to it. This attribute has the following form:

switch_is(*attr_var*)

where *attr_var* is the name of the parameter or structure field that is the discriminator for the union.

If a nonencapsulated union is used as a structure field, the discriminator of the union must be a field of the same structure. If a nonencapsulated union is used as a parameter of an operation, the discriminator must be another parameter of the same operation.

The following example shows uses of a nonencapsulated union.

```
typedef
 [switch_type(long)] union {
  [case (1,3)] float a_float;
  [case (2)] short b_short;
  [default] ; /* An empty arm. Nothing is shipped. */
 } n_e_union_t;

typedef
 struct {
  long a; /* The discriminant for the  */
  /* union later in this struct. */
  [switch_is (a)] n_e_union_t b;
 } a_struct;
```

```
/* Note switch can follow union in operation */
void op1 (
  [in] handle_t h,
  [in,switch_is (s)] n_e_union_t u,
  [in] long s
);
```

Enumeration

An IDL enumeration provides names for integers. It is specified as follows:

```
enum { identifier[= integer], ...}
```

Each identifier in an enumeration is assigned an integer, either explicitly in the interface or automatically by the IDL compiler. If all the identifiers are unassigned, the IDL compiler begins assigning 0 (zero) to the first identifier, 1 to the next identifier, and so on. If an unassigned identifier follows an assigned one, the compiler restarts number assignment with the next consecutive integer. An enumeration can have up to 32,767 identifiers.

Assignments can be made in any order, and multiple identifiers can have the same value. For example:

```
typedef enum {
SHOVEL = 9, AX, MATTOCK = 3, PITCHFORK, SPADE = 9
} yard_tools;
/* values assigned: SHOVEL:9, AX:10, MATTOCK:3, PITCHFORK:4, */
/* SPADE:9
*/
```

Pipes

IDL supports pipes as a mechanism for transferring large quantities of typed data. An IDL pipe is an open-ended sequence of elements of one type. A pipe permits application-level optimization of bulk data transfer by allowing the overlap of communication and processing. Applications that process a stream of data as it arrives, rather than simply storing the data in memory, can make efficient use of the pipe mechanism.

A pipe is specified as follows:

```
pipe type_specifier
```

The *type_specifier* specifies the type for the elements of the pipe. This type cannot be a pointer type, a type that contains a pointer, a conformant type, a context handle, a **handle_t** element type, or a data type that is declared as **transmit_as**.

A pipe type can be used to declare only the type of an operation parameter. IDL recognizes three kinds of pipes, based on the three operation parameters:

- An **in** pipe is for transferring data from a client to a server. It allows the callee (server) to pull an open-ended stream of typed data from the caller (client).
- An **out** pipe is for transferring data from a server to a client. It allows the callee (server) to push the stream of data to the caller (client).
- An **in,out** pipe provides for two-way data transfer between a client and server by combining the behavior of **in** and **out** pipes.

A pipe can be defined only through a **typedef** declaration. Anonymous pipe types are not supported.

At the interface between the stub and the application-specific code (for both the client and server), a pipe appears as a simple callback mechanism. To the user code, the processing of a pipe parameter appears to be synchronous. The IDL implementation of pipes in the RPC stub and runtime allows the apparent callbacks to occur without requiring actual remote callbacks. As a result, pipes provide an efficient transfer mechanism for large amounts of data.

Note however, that pipe data communications occur at about the same speed as arrays. Pipes can improve latency and minimum memory utilization, but not throughput. Pipes are intended for use where the receiver can process the data in some way as it arrives, for example by writing it to a file or passing it to a consumer thread. If the intent is to store the data in memory for later processing, pipes offer no advantage over arrays.

IDL Pipes Example

To illustrate the IDL implementation of pipes, consider the following IDL fragment:

```
typedef
  pipe element_t pipe_t;
```

When the code containing this fragment is compiled, the IDL compiler will generate the following declarations in the derived header file:

```
typedef struct pipe_t {
  void (* pull)(
    rpc_ss_pipe_state_t state,
    element_t *buf,
    idl_ulong_int esize,
    idl_ulong_int *ecount
  );
  void (* push)(
   rpc_ss_pipe_state_t state,
   element_t *buf,
   idl_ulong_int ecount
  );
  void (* alloc)(
   rpc_ss_pipe_state_t state,
   idl_ulong_int bsize,
   element_t **buf,
   idl_ulong_int *bcount
  );
  rpc_ss_pipe_state_t state;
} pipe_t;
```

The pipe data structure specifies pointers to three separate routines and a pipe state. The client application has to implement these routines for the client stub to call, and the server manager must call the associated routines generated in the server stub.

The **pull** routine is used for an input pipe. It pulls the next chunk of data from the client application into the pipe. The input parameters include the pipe **state**, the buffer (***buf**) containing a chunk of data, and the size of the buffer (**esize**) in terms of the number of pipe data elements. The output parameter is the actual count (***ecount**) of the number of pipe data elements in the buffer.

The **push** routine is used for an output pipe. It pushes the next chunk of data from the pipe to the client application. The input parameters include the pipe **state**, the buffer (***buf**) containing a chunk of data, and a count (**ecount**) of the number of pipe data elements in the buffer.

The **alloc** routine allocates a buffer for the pipe data. The input parameters include the pipe **state** and the requested size of the buffer (**bsize**) in bytes. The output parameters include a pointer to the allocated buffer (****buf**), and the actual count (**bcount**) of the number of bytes in the buffer. The routine allocates memory from which pipe data can be marshalled or into which pipe data can be marshalled. If less memory is allocated than requested, the RPC runtime uses the smaller memory and makes more callbacks to the user. If the routine allocates more memory than requested, the excess memory is not used.

Finally, the **state** is used to coordinate between these routines.

For more on how to write the code for the client and server manager, see "Chapter 17. Topics in RPC Application Development" on page 313.

Rules for Using Pipes

Observe the following rules when defining pipes in IDL:

- Pipe types must only be parameters. In other words, pipes of pipes, arrays of pipes, and structures or unions containing pipes as members are illegal.
- A pipe cannot be a function result.
- The element type of a pipe cannot be a pointer or contain a pointer.
- The element type of a pipe cannot be a **context_handle** or **handle_t** type.
- A pipe type cannot be used in the definition of another type. For example, the following code fragment is illegal:

```
typedef
  pipe char pipe_t;

typedef
  pipe_t * pipe_p_t;
```

- A pipe type cannot have the **transmit_as** attribute.
- The element type of a pipe cannot have the **transmit_as** attribute.
- A pipe parameter can be passed by value or by reference. A pipe that is passed by reference (that is, has an * (asterisk)) cannot have the **ptr** or **unique** parameter attributes.
- Pipes that pass data from the client to the server must be processed in the order in which they occur in an operation's signature. All such pipes must be processed before data is sent from the server to the client.
- Pipes that pass data from the server to the client must be processed in the order in which they occur in an operation's signature. No such pipes must be processed until all data has been sent from the client to the server.
- Manager routines must reraise RPC pipe and communications exceptions so that client stub code and server stub code continue to execute properly.

  For example, consider an interface that has an **out** pipe along with other **out** parameters. Suppose that the following sequence of events occurs:

  – The manager routine closes the pipe by writing an empty chunk whose length is 0 (zero).

  – The manager routine attempts to write another chunk of data to the pipe.

  – The generated **push** routine raises the exception **rpc_x_fault_pipe_closed**.
  – The manager routine catches the exception and does not reraise it.
  – The manager routine exits normally.
  – The server stub attempts to marshall the **out** parameters.

  After this sequence, neither the server stub nor the client stub can continue to execute properly.

  To avoid this situation, you *must* reraise the exception.
- A pipe cannot be used in an operation that has the **broadcast** or **idempotent** attribute.
- The element type of a pipe cannot be a conformant structure.
- The maximum length of pipe type IDs is 29 characters.

Arrays

IDL supports the following types of arrays:
- Fixed: The size of the array is defined in IDL and all of the data in the array is transferred during the call.
- Conformant: The size of the array is determined at runtime. At least one bound of the array is determined at runtime by a value referenced through a **min_is**, **max_is**, or **size_is** attribute. All of the data in the array is transferred during the call.
- Varying: The size of the array is defined in IDL but the part of its contents transferred during the call is determined by the values of fields or parameters named in one or more data limit attributes. The data limit attributes are **first_is**, **length_is**, and **last_is**.

An array can also be both conformant and varying (or, as it is sometimes termed, open). In this case, the size of the array is determined at runtime by the value of the field or parameter referenced by the **min_is**, **max_is** or **size_is** attributes. The part of its contents transferred during the call is determined by the values of fields or parameters named in one or more of the data limit attributes.

An IDL array is declared via an *array_declarator* construct whose syntax is as follows:

```
array_identifier array_bounds_declarator ...
```

An *array_bounds_declarator* must be specified for each dimension of an array.

Array Bounds

The *array_bounds_declarator* for the first dimension of an array can take any of the following forms:

[*lower***..** *upper***]**
: The lower bound is *lower*. The upper bound is *upper*.

[*size***]**
: The lower bound is 0 (zero). The upper bound is *size* − 1.

[*]
: The lower bound is 0 (zero). The upper bound is determined by a **max_is** or **size_is** attribute.

[ ]
: The lower bound is 0 (zero). The upper bound is determined by a **max_is** or **size_is** attribute.

[*lower* **..]**
The lower bound is *lower*. The upper bound is determined by a **max_is** or **size_is** attribute.

[* .. *upper***]**
The lower bound is determined by a **min_is** attribute. The upper bound is *upper*.

[* .. *] The lower bound is determined by a **min_is** attribute. The upper bound is determined by a **size_is** or **max_is** attribute.

Conformance in Dimensions Other Than the First

If a multidimensional array is conformant in a dimension other than the first, the C description for this array, which is located in the header (**.h**) file generated by the IDL compiler, will be a one-dimensional conformant array of the appropriate element type. This occurs because there is no "natural" C binding for conformance in dimensions other than the first.

The following examples show how IDL type definitions and parameter declarations that contain bounds in dimensions other than the first are translated into their C equivalents at runtime.

IDL Type Definition:

```
typedef struct {
  long a;
  long e;
  [max_is(,,e),min_is(a)] long g7[*..1][2..9][3..*];
} t3;
```

C Translation:

```
typedef struct {
 idl_long_int a;
 idl_long_int e;
 idl_long_int g7[1];
```

IDL Parameter Declaration:

```
[in,out,max_is(,,e),min_is(a)] long g7[*..1][2..9][3..*];
```

C Translation:

```
/* [in, out] */ idl_long_int g7[]
```

Arrays that have a nonzero first lower bound and a first upper bound that is determined at runtime are translated into the equivalent C representation of a conformant array, as shown in the following IDL type definition and parameter declaration examples:

IDL Type Definition:

```
typedef struct {
      long s;
      [size_is(s)] long fa3[3..*][-4..1][-1..2];
} t1;
```

C Translation:

```
typedef struct {
 idl_long_int s;
 idl_long_int fa3[1][6][4];
} t1;
```

IDL Parameter Declaration:

```
[in,out,size_is(s)] long fa3[3..*][-4..1][-1..2]
```

C Translation:

```
/* [in, out] */ idl_long_int fa3[][6][4]
```

Array Attributes

Array attributes specify the size of an array or the part of an array that is to be transferred during a call. An array attribute specifies a variable that is either a field in the same structure as the array or a parameter in the same operation as the array.

An *array_attribute* can take the following forms:

min_is (*[*] variable*)
max_is (*[*] variable*)
size_is (*[*] variable*)
last_is (*[*] variable*)
first_is (*[*] variable*)
length_is (*[*] variable*)

where *variable* specifies a variable whose value at runtime will determine the bound or element count for the associated dimension. A pointer variable is indicated by preceding the variable name with an ***** (asterisk).

If the array is a member of a structure, any referenced variables must be members of the same structure. If the array is a parameter of an operation, any referenced variables must be parameters of the same operation.

Only the **..._is(** *variable***)** form is allowed when the array is a field of a structure. In this case, the **..._is(*** *variable***)** form is not allowed.

Note that an array with an array attribute (that is, a conformant or varying array) is not allowed to have the **transmit_as** attribute.

The min_is Attribute: The **min_is** attribute is used to specify the variable(s) from which the values of one or more lower bounds of the array will be obtained at runtime. If any dimension of an array has an unspecified lower bound, the array must have a **min_is** attribute. A variable must be identified for each such dimension. The following examples show the syntax of the **min_is** attribute:

```
/* Assume values of variables are as
follows
  long a = -10;
  long b = -20;
  long c = -30;
*/

long [min_is(a)] g1[*..10];      /* g1[-10..10]     */
long [min_is(a)] g2[*..10][4];    /* g2[-10..10[0..3]   */
long [min_is(a,b)] g3[*..10][*..20]; /* g3[-10..10][-20..20] */
long [min_is(,b)] g4[2][*..20];    /* g4[0..1][-20..20]  */
long [min_is(a,,c)] g5[*..7][2..9][*..8];
                /* g5[-10..7][2..9][-30..8]  */
long [min_is(a,b,)] g6[*..10][*..20][3..8];
               /* g6[-10..10][-20..20][3..8]
*/
```

The following examples show the **min_is** attribute being applied to the first dimension of an array in an IDL type definition and parameter declaration, and how the definition or parameter is translated into its C equivalent:

IDL Type Definition:

```
typedef struct {
     long n;
 [min_is(n)] long fa3[*..10][-4..1][-1..2]
} t2;
```

C Translation:

```
typedef struct {
 idl_long_int n;
 idl_long_int fa3[1][6][4];
} t2;
```

IDL Parameter Declaration:

```
[in,out,min_is(n)] long fa3[*..10][-4..1][-1..2]
```

C Translation:

```
/* [in, out] */ idl_long_int fa3[][6][4]
```

The max_is Attribute: The **max_is** attribute is used to specify the variables from which the values of one or more upper bounds of the array are obtained at runtime. If any dimension of an array has an unspecified upper bound, the array must have a **max_is** or **size_is** attribute. A variable must be identified for each dimension in which the upper bound is unspecified. In a **max_is** attribute, the value in the identified variable specifies the maximum array index in that dimension. An array with one or more unspecified upper bounds may have a **max_is** attribute or a **size_is** attribute, but not both.

The **max_is** attribute is for use with conformant arrays. The following is an example of the **max_is** attribute:

```
/* Assume values of variables are as follows:
  long a = 10;
  long b = 20;
  long c = 30;
*/

long [max_is(a)] f1[];      /* f1[0..10] /*
long [max_is(a)] f2[][4];    /* f2[0..10][0..3] */
long [max_is(a,b)] f3[][];    /* f3[0..10][0..20] */
long [max_is(,b)] f4[2][];    /* f4[0..1][0..20] */
long [max_is(a,,c)] f5[1..*][2..9][3..*]; /* f5[1..10][2..9][3..30] */
long [max_is(a,b,)] f6[1..*][2..*][3..8]; /* f6[1..10][2..20][3..8] */
```

The size_is Attribute: The **size_is** attribute is used to specify the variables from which the values of the element counts for one or more dimensions of the array are obtained at runtime. If any dimension of an array has an unspecified upper bound, the array must have a **max_is** or **size_is** attribute. A variable must be identified for each dimension in which the upper bound is unspecified. In a **size_is** attribute, the value in the identified variable specifies the number of elements in that dimension. An array with one or more unspecified upper bounds may have a **max_is** attribute or a **size_is** attribute, but not both.

The size of a dimension is defined as the upper bound, minus the lower bound, + 1.

The **size_is** attribute is for use with conformant arrays. The following is an example of the **size_is** attribute:

```
/* Assume the following values for the referenced
variables:
  n3 = 5;
  x2 = 12;
  x3 = 14;
  z2 = 9;
  z3 = 10;
*/

/* The following declaration */

int [min_is(,,n3),max_is(,x2,x3)] hh[3..13,4..*,*..*];

/* specifies the same data to be */
/* transmitted as the declaration */

int [min_is(,,n3),size_is(,z2,z3)]
hh[3..13,4..*,*..*];
```

The last_is Attribute: The **last_is** attribute is one of the attributes that can be used to allow the amount of data in an array that will be transmitted to be determined at runtime. Each **last_is** attribute specifies an upper data limit, which is the highest index value in that dimension for the array elements to be transmitted. If the entry in a **last_is** attribute for a dimension is empty, the effect is as if the upper bound in that dimension had been specified.

An array can have either the **last_is** attribute or the **length_is** attribute, but not both.

When an array with the **last_is** attribute is used in a remote procedure call, the elements actually passed in the call can be a subset of the maximum possible.

The **last_is** attribute is for use with varying arrays. The following is an example of the **last_is** attribute:

```
/* Assume the following values for the
referenced variables:
  long a = 1;
  long b = 2;
  long c = 3;
  long e = 25;
  long f = 35;
*/

long [last_is(a,b)] bb1[10][20]; /* transmit bb1[0..1][0..2] */
long [last_is(a,b)] bb2[-1..10][-2..20][-3..30];
                 /* transmit bb2[-1..1][-2..2][-3..30] */
long [last_is(a,,c)] bb3[-1..10][-2..20][-3..30];
                 /* transmit bb3[-1..1][-2..20][-3..3] */
long [last_is(,b,c),max_is(,e)] cc1[10][][30];
```

```
                /* transmit cc1[0..9][0..2][0..3] */
long [last_is(a,b),max_is(,e,f)] cc2[-4..4][][];
                /* transmit cc2[-4..1][0..2][0..35] */
```

The first_is Attribute: The **first_is** attribute is one of the attributes that can be used to allow the amount of data in an array that will be transmitted to be determined at runtime. Each **first_is** attribute specifies a lower data limit, which is the lowest index value in that dimension for the array elements to be transmitted. If the entry in a **first_is** attribute for a dimension is empty, the effect is as if the lower bound in that dimension had been specified.

When an array with the **first_is** attribute is used in a remote procedure call, the elements actually passed in the call can be a subset of the maximum possible.

The **first_is** attribute is for use with varying arrays. The following is an example of the **first_is** attribute:

```
/* Assume the following values for the
referenced variables:
  long p = -1;
  long q = -2;
  long r = -3;
  long t = -25;
  long u = -35;
  long x = 1;
  long y = 2;
  long z = 3;
*/

long [first_is(p)] dd1[-10..10];       /* transmit dd1[-1..10] */
long [first_is(p),last_is(x)] dd2[-10..10]; /* transmit dd2[-1..1] */
long [first_is(p,q)] ee1[-10..10][-20..20];
             /* transmit ee1[-1..10][-2..20] */
long [first_is(p,q)] ee2[-10..10][-20..20][-30..30];
             /* transmit ee2[-1..10][-2..20][-30..30] */
long [first_is(p,q,r),last_is(,,z)] ee3[-10..10][-20..20][-30..30]:
             /* transmit ee3[-1..10][-2..20[-3..30] */
double [first_is(,q,r),min_is(,t)] ff1[10][*..2][-30..30];
             /* transmit ff1[0..9][-2..2][-3..30] */
double [first_is(p,q),min_is(,t,u)] ff2[-4..4][*..2][*..35];
             /* transmit ff2[-1..4][-2..2][-35..35] */
double [max_is(x,,z),min_is(,t,u),first_is(p,,r)] ff3[-20..*][*..30][*..*]
             /* transmit ff3[-1..1][-25..30][-3..3] */
```

The length_is Attribute: The **length_is** attribute is one of the attributes that can be used to allow the amount of data in an array that will be transmitted to be determined at runtime. Each **length_is** attribute specifies the number of elements in that dimension to be transmitted. If the entry in a **length_is** attribute for a dimension is empty, the effect is for the highest index value in that dimension for the elements to be transmitted to be determined from the upper bound in that dimension.

An array can have either the **last_is** attribute or the **length_is** attribute, but not both.

When an array with the **length_is** attribute is used in a remote procedure call, the elements actually passed in the call can be a subset of the maximum possible.

The **length_is** attribute is for use with varying arrays. The following is an example of the **length_is** attribute:

```
/* Assume the following values for the referenced
variables:
  n3 = 5;
  f2 = 10;
  a1 = 11;
  a2 = 12;
  a3 = 14;
  e1 = 9;
  e2 = 3;
  e3 = 10;
*/
/* The following declaration: */

int [min_is(,,n3),first_is(,f2,),last_is(a1,a2,a3)] \
 gg[3..13,4..14,*..15];

/* specifies the same data to be  */
/* transmitted as the declaration: */

int [min_is(,,n3),first_is(,f2,),length_is(e1,e2,e3)] \
 gg[3..13,4..14,*..15];
```

Rules for Using Arrays

Observe the following rules when defining arrays in IDL:

- A structure can contain only one conformant array, which must be the last member in the structure.
- Conformant arrays are not valid in unions.
- A structure parameter containing a conformant array can be passed only by reference.
- Arrays that have the **transmit_as** attribute cannot be conformant or varying arrays.
- The structure member or parameter referenced in an array attribute cannot be defined to have either the **represent_as** or **transmit_as** attribute.
- Array bounds must be integers. Array attributes can reference only structure members or parameters of integer type.
- A parameter that is referenced by an array attribute on a conformant array must have the **in** attribute.
- Array elements cannot be context handles or pipes, or conformant arrays or conformant structures.

Strings

IDL implements strings as one-dimensional arrays to which the **string** attribute is assigned. The element type of the array must resolve to one of the following:

- Type **char**
- Type **byte**
- A structure all of whose members are of type **byte** or of a named type that resolves to **byte**
- A named type that resolves to one of the previous three types
- Type **unsigned short**
- Type **unsigned long**
- A named type that resolves to **unsigned short** or **unsigned long**

Strings built from **byte** or **char** data types are referred to as byte-string types while strings built from **unsigned short** or **unsigned long** types are called integer-string types. Integer string types allow for multioctet character sets whose characters are represented by 16-bit or 32-bit quantities, rather than as groups of bytes. For example:

```
/* A structure that contains a fixed string */
/* and a conformant string */
typedef unsigned long PRIVATE_CHAR_32;
typedef struct {
  [string] PRIVATE_CHAR_32 fixed[27];
  [string] PRIVATE_CHAR_32 conf[];
} two_strings;

/* A structure that contains pointers to two strings */
typedef unsigned short PRIVATE_CHAR_16;
typedef struct {
  [string] PRIVATE_CHAR_16 *astring;
  [string] PRIVATE_CHAR_16 *bstring;
} stringptrs;
```

Integer-string types use the array element zero (0) to specify the string terminator, while byte-string types use the NULL character. Both byte-type and integer-type strings conform to the same usage rules.

An array with the **string** attribute represents a string of characters. The **string** attribute does not specify the format of the string or the mechanism for determining its length. Implementations of IDL provide string formats and mechanisms for determining string lengths that are compatible with the programming languages in which applications are written. For DCE RPC IDL, the number of characters in a **string** array includes the NULL terminator (for byte-string types) or the zero (0) terminator (for integer-string types), and the entire terminated string is passed between stubs.

The *array_bounds_declarator* for a **string** array determines the maximum number of characters in the array. Note that, when you declare a string, you must allocate space for one more than the maximum number of characters the string is to hold. For instance, if a string is to store 80 characters, the string must be declared with a size of 81:

```
/* A string type that holds 80 characters */
typedef
  [string] char string_t [81];
```

If an array has the **string** attribute or if the type of an array has the **string** attribute, the array cannot have the **first_is**, the **last_is**, or the **length_is** attribute.

Pointers

Use the following syntax to declare an IDL pointer:

```
*[*]...pointer_identifier
```

The ***** (asterisk) is the pointer operator, and multiple asterisks indicate multiple levels of indirection.

Pointer Attributes

Pointers are used for several purposes, including implementing a parameter passing mechanism that allows a data value to be returned, and building complex data structures.

IDL offers three classes of pointers: reference pointers, full pointers, and unique pointers. The attributes that indicate these pointers are as follows:

- **ref**: Indicates reference pointers. This is the default for top-level pointers used in parameters.
- **ptr**: Indicates full pointers.
- **unique**: Indicates unique pointers.

Pointer attributes are used in parameters, in structure and union members, and in type definitions. In some instances, IDL infers the applicable pointer class from its usage. However, most pointer declarations require that you specify a pointer class by using one of the following methods:

- Use the **ref**, **ptr**, or **unique** attribute in the pointer declaration.
- Use the **pointer_default** attribute in the IDL interface heading. The default pointer class is determined by the **pointer_default** attribute.

Pointer attributes are applied only to the top-level pointer within the declaration. If multiple pointers are declared in a single declaration, the **pointer_default** established applies to all but the top-level pointer. (See "Pointer Attributes in Parameters" on page 394, which describes pointer attributes in parameters.)

Examples of pointers are shown at the end of this section.

Reference Pointers: A reference pointer is the least complex form of pointer. The most common use for this class of pointer is as a passing mechanism; for example, passing an integer by reference. Reference pointers have significantly better performance than full pointers, but are restrictive; you cannot create a linked list by using a reference pointer because a reference pointer cannot have a NULL value, and the list cannot be terminated.

A reference pointer has the following characteristics:

- It always points to valid storage; it can never have a NULL value.
- Its value does not change during a call; it always points to the same storage on return from the call as it did when the call was made.
- It does not support aliasing; it cannot point to a storage area that is pointed to by any other pointer used in a parameter of the same operation.

When a manager routine is entered, all the reference pointers in its parameters will point to valid storage, except those reference pointers that point neither to targets whose size can be determined at compile time nor to values that have been received from the client.

In the following example, the size of the targets of the reference pointers can be calculated at compilation time:

```
typedef [ref] long *rpl;

void op1([in] long f,
     [in] long l,
     [in,first_is(f),last_is(l)] rpl rpla[10]);
```

For this example, when the manager is entered, all the pointers in **rpla** will point to usable storage, although only ***rpla[f]** through ***rpla[l]** will be the values received from the client.

Conversely, the size of the targets of the reference pointers cannot be calculated at compile time in the following example:

```
typedef [ref,string] char *rps;

void op1([in] long f,
     [in] long l,
     [in,first_is(f),last_is(l)] rps rpsa[10]);
```

In this case, only **rpsa[f]** through **rpsa[l]**, which point to values received from the client, will point to usable storage.

Full Pointers: A full pointer is the most complex form of pointer. It supports all capabilities associated with pointers. For example, by using a full pointer you can build complex data structures such as linked lists, trees, queues, or arbitrary graphs.

A full pointer has the following characteristics:

- Its value can change during a call; it can change from a NULL to non-NULL value, non-NULL to NULL, or from one non-NULL value to another non-NULL value.
- It supports aliasing; it can point to a storage area that is also pointed to by any other full pointer used in a parameter of the same operation. However, all such pointers must point to the beginning of the structure. There is no support for pointers to substructures or to overlapping storage areas. For example, if the interface definition code contains the following:

  ```
  [uuid(0e256080-587c-11ca-878c-08002b111685), version(1.0)]
  interface overlap
  {
   typedef struct {
       long bill;
       long charlie;
   } foo;
   typedef struct {
       long fred;
       foo ken;
   } bar;

   void op ([in] foo *f, [in] bar *b);
  }
  ```

  and the client application code includes the following:

  ```
  bar bb;
   .
   .
   .
  op (&bb.ken, &bb);
  ```

  then the server stub treats these two separate parameters as distinct, and the manager application code does not see them as overlapping storage.
- It allows dynamically allocated data to be returned from a call.

Note that you might need to take some extra steps if you use large linked lists in your application. Linked lists are marshalled and unmarshalled using recursion which can cause the stack size to grow. Linked lists usually do not cause problems in simple clients that do not spawn threads for remote procedure calls. In this case, the stack can grow as needed.

Large linked lists can cause problems in servers because the server's thread-stack usually cannot grow automatically. Large lists can overrun the stack, causing the server to crash.

DCE offers several ways to avoid this server memory problem while using large linked lists.

One method is to increase the server stack size using the **rpc_mgmt_set_server_stack_size()** routine. This method is useful when you suspect that the linked list is just slightly larger than the server stack. For information about using the **rpc_mgmt_set_server_stack_size()** routine, refer to the OSF DCE Application Development Reference.

If you suspect that the list size is much greater than the stack, you can convert the list to an array using the **transmit_as** idl attribute. Servers handle arrays by allocating memory from the heap rather than from the stack. For information about using the **transmit_as** idl attribute, refer to "Chapter 19. Attribute Configuration Language" on page 425 in this guide.

Unique Pointers: A unique pointer is more flexible than a reference pointer. However, both types of pointers share several important characteristics.

A unique pointer has the following characteristics:

- It can have a NULL value.
- It can change from NULL to non-NULL during a call. This change results in memory being allocated on return from the call, whereby the result is stored in the allocated memory.
- It can change from non-NULL to NULL during a call. This change can result in the orphaning of the memory pointed to on return from the call. Note that, if a unique pointer changes from one non-NULL value to another non-NULL value, the change is ignored.
- It does not identify particular extents of memory, but only extents of memory that are suitable for storing the data. If it is important to know that the data is being stored in a specific memory location, then you should use a full pointer.
- If it has a value other than NULL, output data is placed in existing storage.

Unique pointers are similar to reference pointers in the following ways:

- No storage pointed to by a unique pointer can be reached from any other name in the operation. That is, a unique pointer does not allow aliasing of data within the operation.
- Data returned from the called subroutine is written into the existing storage specified by the unique pointer, if the pointer did not have the value NULL.

With regard to performance, unique pointers have an advantage over full pointers because unique pointers do not support the referencing of common data by more than one pointer (aliasing), and they are significantly more flexible than reference pointers because they can have a value of NULL.

Unique pointers are particularly suitable for creating optional parameters (because you can specify them as NULL) and for simple tree or singly linked-list data structures. You specify the three different levels of pointers by attributes, as follows:

[ref] Reference pointers

[unique]
Unique pointers

[ptr] Full pointers

The following example shows how a unique pointer can be used:

```
[
  uuid(D37A0E80-5D23-11C9-B199-08002B13D56D)
] interface Unique_ptrs
{
  typedef [ref]  long *r_ptr;
  typedef [unique] long *u_ptr;
  typedef [ptr]  long *f_ptr;

  void op1 (
   [ref,in,out,string]  char *my_rname,
   [unique,in,out,string] char *my_uname,
   [ptr,in,out,string]  char *my_pname
    );
}
```

Pointer Attributes in Parameters

A pointer attribute can be applied to a parameter only if the parameter contains an explicit pointer declaration (*****).

By default, a single pointer (*****) operator in a parameter list of an operation declaration is treated as a reference pointer. To override this, specify a pointer attribute for the parameter. When there is more than one pointer operator, or multiple levels of indirection in the parameter list, the rightmost pointer is the top-level pointer; all pointers to the left of the rightmost pointer are of a lower level. The top-level pointer is treated as a reference pointer by default; the lower-level pointers have the semantics specified by the **pointer_default** attribute in the interface.

The following example illustrates the use of top- and lower-level pointers:

```
void op1 ([in] long **p_p_l)
```

In this example, **p_p_l** is a pointer to a pointer to a long integer. The first or leftmost pointer (*****) signifies that the pointer to the long integer is a lower-level pointer, and the second or rightmost pointer (*) signifies that the pointer to the pointer is a top-level pointer.

Any pointer attribute you specify for the parameter applies to the top-level pointer only. Note that unless you specify a pointer attribute, the top-level explicit pointer declaration in a parameter defaults to a reference pointer even if the **pointer_default(ptr)** interface attribute is specified.

Using a reference pointer improves performance but is more restrictive. For example, the pointer declared in the following operation, for the parameter **int_value**, is a reference pointer. An application call to this operation can never specify NULL as the value of **int_value**.

```
void op ([in] long *int_value);
```

To pass a NULL value, use a full pointer. The following two methods make **int_value** into a full pointer:

- Applying the **ptr** attribute to the declaration of the parameter, **int_value**:

  ```
  void op ([in, ptr] long *int_value);
  ```
- Using the **pointer_default (ptr)** attribute in an interface header :

  ```
  [uuid(135e7f00-1682-11ca-bf61-08002b111685,
   pointer_default(ptr),
   version(1.0)] interface full_pointer
  {
  typedef long *long_ptr;
  void op ([in] long_ptr int_value);
  }
  ```

A NULL pointer can also be passed via a unique pointer.

Array Attributes on Pointers

To apply array attributes to pointers, use the **max_is** or **size_is** attributes. When applied to a pointer, the **max_is** and **size_is** attributes convert the pointer from a single element of a certain type to a pointer to an array of elements of that type. The number of elements in the array is determined by the variable in the **max_is** and **size_is** attributes.

Pointer Attributes in Function Results

Function results that are pointers are always treated as full pointers. The **ptr** attribute is allowed on function results but it is not mandatory. The **ref** pointer attribute is never allowed on function results.

A function result that is a pointer always indicates new storage. A pointer parameter can reference storage that was allocated before the function was called, but a function result cannot.

Pointers in Structure Fields and Union Case

If a pointer is declared in a member of a structure or union, its default is determined by the **pointer_default** attribute you specify for the interface. To override this, specify a pointer attribute for the member.

Resolving a Possible Pointer Ambiguity

A declaration of the following form raises a possible ambiguity about the type of *myarray*:

```
void op ([in, out] long s, [in, out, size_is(s)] long **myarray);
```

IDL defines *myarray* in this case to be an array of pointers to **longs**, not a pointer to an array of **longs**. The **max_is** and **size_is** attributes always apply to the top-level, or rightmost, ***** (asterisk) in the IDL signature of a parameter.

Rules for Using Pointers

Use the following rules when developing code in IDL:

- Do not use the full pointer attribute on the following:
  - The parameter in the first parameter position, when that parameter is of type **handle_t** or is of a type with the **handle** attribute.
  - Context handle parameters.
  - A parameter that has the output attribute (**out**), but not the input attribute (**in**).

- The element type of a pipe must not be a pointer or a structure containing a pointer.
- A member of a union or a structure contained in a union cannot contain a reference pointer.
- A reference pointer must point to valid storage at the time the call is made.
- A parameter containing a varying array of reference pointers must have all array elements initialized to point to valid storage even if only a portion of the array is input, since the manager code (the application code supporting an interface on a server) may use the remaining array elements. (Recall that a varying array is one to which any of the array attributes **first_is, last_is, length_is** is applied).
- The type name in a declaration that defines a pointer type must have no more than 28 characters.

Memory Management for Pointed-to Nodes

A full pointer can change its value across a call. Therefore, stubs must be able to manage memory for the pointed-to nodes. Managing memory involves allocating and freeing memory for user data structures.

Allocating and Freeing Memory: Manager code within RPC servers usually uses the **rpc_ss_allocate()** routine to allocate storage. Storage that is allocated by **rpc_ss_allocate()** is released by the server stub after any output parameters have been marshalled by the stubs. Storage allocated by other allocators is not released automatically but must be freed by the manager code. When the manager code makes a remote call, the default memory management routines are **rpc_ss_allocate()** and **rpc_ss_free()**.

The syntax of the **rpc_ss_allocate()** routine is as follows:

```
idl_void_p_t rpc_ss_allocate (idl_size_t size);
```

The *size* parameter specifies the size of the memory allocated.

Note: In ANSI standard C environments, **idl_void_p_t** is defined as **void *** and in other environments is defined as **char ***.

Use **rpc_ss_free()** to release storage allocated by **rpc_ss_allocate()**. You can also use the **rpc_ss_free()** routine to release storage pointed to by a full pointer in an input parameter and have the freeing of the memory reflected on return to the calling application by specifying the **reflect_deletions** attribute as an *operation_attribute*. See “Declarations” on page 359 for more information.

The syntax of the routine is as follows:

```
void rpc_ss_free (idl_void_p_t node_to_free);
```

The *node_to_free* parameter specifies the location of the memory to be freed.

Enabling and Disabling Memory Allocation: It may be necessary to call manager routines from different environments; for example, when the application is both a client and a server of the same interface. In this case, the same routine may be called both from server manager code and from client code. The **rpc_ss_allocate()** routine, when used by the manager code to allocate memory, must be initialized before its first use. The stub performs the initialization automatically. Code, other than stub code, that calls a routine, which in turn calls **rpc_ss_allocate()**, first calls the **rpc_ss_enable_allocate()** routine.

The syntax of the routine is as follows:

```
void rpc_ss_enable_allocate (void);
```

The environment set up by the **rpc_ss_enable_allocate()** routine is released by calling the **rpc_ss_disable_allocate()** routine. This routine releases all memory allocated by calls to **rpc_ss_allocate()** since the call to **rpc_ss_enable_allocate()** was made. It also releases memory that was used by the memory management mechanism for internal bookkeeping.

The syntax of the **rpc_ss_disable_allocate()** routine is as follows:

```
void rpc_ss_disable_allocate (void);
```

Advanced Memory Management Support

Memory management may also involve setting and swapping the mechanisms used for allocating and freeing memory. The default memory management routines are **malloc()** and **free()**, except when the remote call occurs within manager code, in which case the default memory management routines are **rpc_ss_allocate()** and **rpc_ss_free()**.

Setting the Client Memory Mechanism: Use the **rpc_ss_set_client_alloc_free()** routine to establish the routines used in allocating and freeing memory.

The syntax of the routine is as follows:

```
void rpc_ss_set_client_alloc_free (
   idl_void_p_t (*p_allocate) (
    idl_size_t size),
   void (*p_free) (
     idl_void_p_t ptr)
   );
```

The *p_allocate* parameter points to a routine that has the same procedure declaration as the **malloc()** routine, and is used by the client stub when performing memory allocation. The *p_free* parameter points to a routine that has the same procedure declaration as the **free()** routine, and is used by the client stub to free memory.

Swapping Client Memory Mechanisms: This routine exchanges the current client allocation and freeing mechanism for one supplied in the call. The primary purpose of this routine is to simplify the writing of modular routine libraries in which RPC calls are made. To preserve modularity, any dynamically allocated memory returned by a modular routine library must be allocated with a specific memory allocator. When dynamically allocated memory is returned by an RPC call that is then returned to the user of the routine library, use **rpc_ss_swap_client_alloc_free()**, before making the RPC call, to make sure the desired memory allocator is used. Prior to returning, the modular routine library calls **rpc_ss_set_client_alloc_free()** to restore the previous memory management mechanism.

The syntax of the routine is as follows:

```
void rpc_ss_swap_client_alloc_free (
   idl_void_p_t (*p_allocate) (
     idl_size_t size),
   void (*p_free) (
     idl_void_p_t ptr),
   idl_void_p_t (**p_p_old_allocate) (
     idl_size_t size),
   void (**p_p_old_free) (
     idl_void_p_t ptr)
   );
```

The *p_allocate* parameter points to a routine that has the same procedure declaration as the **malloc()** routine, and is used by the client stub when performing memory allocation. The *p_free* parameter points to a routine that has the same procedure declaration as the **free()** routine, and is used by the client stub to free memory. The *p_p_old_allocate* parameter points to a pointer to a routine that has the same procedure declaration as the **malloc()** routine, and is the default routine used for memory allocation in the client stub. The *p_p_old_free* parameter points to a pointer to a routine that has the same procedure declaration as the **free()** routine, and is used for memory release in the client.

Use of Thread Handles in Memory Management

There are two situations where control of memory management requires the use of thread handles. The more common situation is when the manager thread spawns additional threads. The less common situation is when a program transitions from being a client to being a server, then reverts to being a client.

Spawning Threads: When a remote procedure call invokes the manager code, the manager code may wish to spawn additional threads to complete the task for which it was called. To spawn additional threads that are able to perform memory management, the manager code must first call the **rpc_ss_get_thread_handle()** routine to get its thread handle and then pass that thread handle to each spawned thread. Each spawned thread that uses the **rpc_ss_allocate()** and **rpc_ss_free()** routines for memory management first calls the **rpc_ss_set_thread_handle()** routine by using the handle obtained by the original manager thread.

These routine calls allow the manager and its spawned threads to share a common memory management environment. This common environment enables memory allocated by the spawned threads to be used in returned parameters, and causes all allocations in the common memory management environment to be released when the manager thread returns to the server stub.

The main manager thread must not return control to the server stub before all the threads it spawned complete execution; otherwise, unpredictable results may occur.

The listener thread can cancel the main manager thread if the remote procedure call is orphaned or if a cancellation occurs on the client side of the application. You should code the main manager thread to terminate any spawned threads before it exits. The code should anticipate exits caused by an unexpected exception or by being canceled.

Your code can handle all of these cases by including a **TRY/FINALLY** block to clean up any spawned threads if a cancellation or other exception occurs. If unexpected exceptions do not concern you, then your code can perform two steps. They are disabling cancelability before threads are spawned followed by enabling cancelability after the join operation finishes and after testing for any pending cancel operations. Following this disable/enable sequence prevents routine **pthread_join()** from producing a cancel point in a manager thread that has spawned threads which, in turn, share thread handles with the manager thread.

Transitioning from Client to Server to Client: Immediately before the program changes from a client to a server, it must obtain a handle on its environment as a client by calling **rpc_ss_get_thread_handle()**. When it reverts from a server to a client, it must reestablish the client environment by calling the **rpc_ss_set_thread_handle()** routine, supplying the previously obtained handle as a parameter.

Syntax for Thread Routines: The syntax for the **rpc_ss_get_thread_handle()** routine is as follows:

```
rpc_ss_thread_handle_t rpc_ss_get_thread_handle(void);
```

The syntax for the **rpc_ss_set_thread_handle()** routine is as follows:

```
void rpc_ss_set_thread_handle (
   rpc_ss_thread_handle_t id
   );
```

The **rpc_ss_thread_handle_t()** value identifies the thread to the RPC stub support library. The *id* parameter indicates the thread handle passed to the spawned thread by its creator, or the thread handle returned by the previous call to **rpc_ss_get_thread_handle()**.

Rules for Using the Memory Management Routines

You can use the **rpc_ss_allocate()** routine in the following environments:

- The manager code for an operation that has a full pointer in its argument list
- The manager code for an operation to which the **enable_allocate** ACF attribute is applied
- Code that is not called from a server stub but that has called the **rpc_ss_enable_allocate()** routine
- A thread, spawned by code of any of the previous three types, that has made a call to the **rpc_ss_set_thread_handle()** routine using a thread handle obtained by this code

Examples Using Pointers

The examples in this subsection contain the following files, listed here with the function of each file:

STRING_TREE.IDL
Defines data types and interfaces

CLIENT.C
User of the interface

MANAGER.C
Server code that implements the procedure

SERVER.C
Declares the server; enables the client code to find the interface it needs

STRING_TREE.OUTPUT
Shows the output

The STRING_TREE.IDL Example

```
[uuid(0144d600-2d28-11c9-a812-08002b0ecef1), version(0)]
interface string_tree
{
 /*
  * Maximum length of a string in the tree
  */
 const long int st_c_name_len = 32;

 /*
  * Definition of a node in the tree.
  */
 typedef struct node
 {
   [string] char name[0..st_c_name_len];
   [ptr] struct node *left;
   [ptr] struct node *right;
 } st_node_t;

 /*
  * Operation that prunes the left subtree of the specified
  * tree and returns it as the value.
  */
 st_node_t *st_prune_left (
   [in, out] st_node_t *tree /* root of tree by ref */
   );
}
```

The CLIENT.C Example

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

#include <stdlib.h>

/*
** Routine to print a depiction of the tree
*/
void st_print_tree (tree, indent)
 st_node_t *tree;
 int indent;
{
 int i;
 if (tree == NULL) return;
 for (i = 0; i < indent; i++) printf("  ");
 printf("%s\n",tree->name);
 st_print_tree(tree->left, indent + 1);
 st_print_tree(tree->right, indent + 1);
}
/*
** Create a tree with a few nodes
*/
st_node_t *st_make_tree()
{
 st_node_t *root = (st_node_t *)malloc(sizeof(st_node_t));
 strcpy(root->name,"Root Node");

 /* left subtree node */
 root->left = (st_node_t *)malloc(sizeof(st_node_t));
 strcpy(root->left->name,"Left subtree");

 /* left subtree children */
 root->left->right = NULL;
 root->left->left = (st_node_t *)malloc(sizeof(st_node_t));
 strcpy(root->left->left->name,"Child of left subtree");
 root->left->left->left = NULL;
 root->left->left->right = NULL;
 /* right subtree node */
 root->right = (st_node_t *)malloc(sizeof(st_node_t));
 strcpy(root->right->name,"Right subtree");
 root->right->left = NULL;
 root->right->right = NULL;

 return root;
}

main()
{
 st_node_t *tree;
 st_node_t *subtree;

 /* setup and print original tree */
 tree = st_make_tree();
 printf("Original Tree:\n");
 st_print_tree(tree, 1);

 /* call the prune routine */
 subtree = st_prune_left (tree);

 /* print the resulting trees */
 printf("\nPruned Tree:\n");
 st_print_tree(tree, 1);

 printf("\nPruned subtree:\n");
 st_print_tree(subtree, 1);
 }
```

The MANAGER.C Example

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

/*
** Prune the left subtree of the specified tree and return
** it as the function value.
*/
st_node_t *st_prune_left (tree)
 /* [in,out] */ st_node_t *tree;
{
 st_node_t *left_sub_tree = tree->left;
 tree->left = (st_node_t *)NULL;
 return left_sub_tree;
}
```

The SERVER.C Example

```
#include <stdio.h>
#include "string_tree.h" /* header created by idl compiler */
#define check_error(s, msg) if(s != rpc_s_ok) \
 {fprintf(stderr, "%s", msg); exit(1);}

main ()
{
 unsigned32       status;  /* error status (nbase.h) */
 rpc_binding_vector_p_t binding_vector;
             /* set of binding handles (rpc.h) */

 rpc_server_register_if( /* register interface with RPC runtime */
  string_tree_v0_0_s_ifspec,
           /* interface specification (string_tree.h) */
  NULL,
  NULL,
  &status                 /* error status */
 );
 check_error(status, "Can't register interface\n");

 rpc_server_use_all_protseqs(  /* establish protocol sequences */
  rpc_c_protseq_max_calls_default,
             /* concurrent calls server takes (rpc.h) */
  &status
 );
 check_error(status, "Can't establish protocol sequences\n");
 rpc_server_inq_bindings( /* get set of this server's binding handles
*/
  &binding_vector,
  &status
 );
 check_error(status, "Can't get binding handles\n");
 rpc_ep_register(   /* register addresses in endpoint map database */
  string_tree_v0_0_s_ifspec,       /* interface specification */
  binding_vector,   /* (string_tree.h) the set of binding handles */
  NULL,
  "",
  &status
 );
 check_error(status, "Can't add address to the endpoint database\n");

 rpc_ns_binding_export(         /* establish namespace entry */
  rpc_c_ns_syntax_dce,      /* syntax of the entry name (rpc.h) */
  "string_tree",        /* entry name in directory service */
  &string_tree_v0_0_s_ifspec,       /* interface specification */
  binding_vector,   /* (string_tree.h) the set of binding handles */
  NULL,
  &status
 );
 check_error(status, "Can't export to directory service\n");

 rpc_binding_vector_free(       /* free set of binding handles */
  &binding_vector,
  status
 );
 check_error(status, "Can't free binding handles and vector\n");

 rpc_server_listen(            /* listen for remote calls */
  rpc_c_listen_max_calls_default,
              /* concurrent calls server executes (rpc.h) */
  &status
 );
 check_error(status, "rpc listen failed\n");
}
```

The STRING_TREE.OUTPUT Example

```
Original Tree:
  Root Node
    Left subtree
      Child of left subtree
    Right subtree
Pruned Tree:
  Root Node
    Right subtree
Pruned subtree:
  Left subtree
    Child of left subtree
```

Customized Handles

The **handle** attribute specifies that the type being declared is a user-defined, nonprimitive handle type, and is to be used in place of the predefined primitive handle type **handle_t**. The term customized handle is used to denote a nonprimitive handle.

The following example declares a customized handle type **filehandle_t**, a structure containing the textual representations of a host and a pathname:

```
typedef [handle] struct {
   char host[256];
   char path[1024];
   } filehandle_t;
```

If the handle parameter is the first parameter in the list, then it is a customized handle that is used to determine the binding for the call, and it must have the **in** attribute or the **in,out** attributes. A handle parameter that is not the first parameter in the parameter list need not have the **in** or **in,out** attributes.

Note that a **handle_t** parameter that is the first parameter in the list must not have the **transmit_as** attribute.

To build an application that uses customized handles, you must write custom binding and unbinding routines, and you must link those routines with your application client code. At runtime, each time the client calls an operation that uses a customized handle, the client stub calls the custom binding routine before it sends the remote procedure call request, and the client stub calls the custom unbinding routine after it receives a response.

The following paragraphs specify C prototypes for customized binding and unbinding routines; in these prototypes, *CUSTOM* is the name of the customized handle type.

The custom binding routine *CUSTOM*_**bind** generates a primitive binding handle from a customized handle and returns the primitive binding handle:

```
handle_t CUSTOM_bind (CUSTOM c-handle)
```

The custom unbinding routine *CUSTOM*_**unbind** takes two inputs, a customized handle and the primitive binding handle that was generated from it, and has no outputs:

```
void CUSTOM_unbind (

 CUSTOM c-handle,
 handle_t rpc-handle)
```

A custom unbinding routine typically frees the primitive binding handle and any unneeded resources associated with the customized handle, but it is not required to do anything.

Because the **handle** attribute can occur only in a type declaration, a customized handle must have a named type. Because customized handle type names are used to construct custom binding and unbinding routine names, these names cannot exceed 24 characters.

A customized handle can be coded either in a parameter list as an explicit handle or in an interface header as an implicit handle.

Context Handles

Manager code often maintains state information for a client. A handle to this state information is passed to the client in an output parameter or as an operation result. The client passes the unchanged handle-to-the-state information as an input or input/output parameter of a subsequent manager operation that the client calls to manipulate that data structure. This handle-to-the-state information is called a context handle. A context handle is implemented as an untyped pointer or a pointer to a structure by tag name.

The manager causes the untyped pointer or the structure pointer to point to the state information it will need the next time the client asks the manager to manipulate the context. For the client, the context handle is an opaque pointer (**idl_void_p_t** or an opaque structure tag). The client receives or supplies the context handle by means of the parameter list but does not perform any transformations on it.

The RPC runtime maintains the context handle, providing an association between the client and the address space running the manager and the state information within that address space.

If a manager supports multiple interfaces, and a client obtains a context handle by performing an operation from one of these interfaces, the client can then supply the context handle to an operation from another of these interfaces.

A context handle can only be exchanged between the server process that created it and the client process for which it was created. No other client except the one that obtained the context handle can use it without causing an application error.

The context_handle Attribute

Specify a context handle by one of the following methods:

- Use the **context_handle** attribute on a parameter of type **void ***.
- Use the **context_handle** attribute on a type that is defined as **void ***.
- Use the **context_handle** attribute on a type that is defined as a pointer to a structure by tag name.

For example, in the IDL file, you can define a context handle within a type declaration as follows:

```
typedef [context_handle] void * my_context;
```

or within a parameter declaration as follows:

```
[in, context_handle] void * my_context;
```

You can also define a context handle within a type declaration as a forward reference to a structure type by tag, as follows:

```
typedef [context_handle] struct opaque_struct * opaque_ch_t;
```

Note that you do not need to define the structure type in the IDL file; it is a forward reference to a structure whose definition can be included into the server code, either from a private **.h** file or from a server IDL file. As a result, the structure type is opaque to the client. This method of defining a context handle provides type checking and permits the server code to avoid extensive casting when manipulating the context handle.

A structure type in a context handle type definition must be referenced by tag name and not by type name. So, for example, the first of the following declarations is valid, while the second is not:

```
typedef [context_handle] struct struct_tag * valid_ch_t;
                         /* valid */

typedef [context_handle] struct_type * invalid_ch_t;
                         /* error
*/
```

The following example illustrates context handles defined as untyped pointers and as pointers to structures by tag name.

```
/* A context handle implemented as untyped pointer */
typedef [context_handle] void * void_ch_t;

/* A context handle implemented as a */
/* pointer to a structure by tag name */
typedef [context_handle] struct opaque_struct * opaque_ch_t;

/* Operations using both types of context handles */
void ch_oper(
   [in] void_ch_t v1,
   [in,out] void_ch_t *v2,
   [out] void_ch_t *v3,
   [in] opaque_ch_t *o2,
   [out] opaque_ch_t *o3
);

void_ch_t void_ch_oper ([in] handle_t h);

opaque_ch_t opaque_ch_oper([in] handle_t h);
```

It is possible to define a structure type in a context handle in the IDL file; for example, the following structure definition can either precede or follow the definition of **valid_ch_t** in the example previously shown:

```
typedef struct struct_tag {long l;} struct_type;
```

This practice is not recommended, however, since it violates the opaqueness of the context handle type.

The type name in a context handle declaration must be no longer than 23 characters.

The first operation on a context creates a context handle that the server procedure passes to the client. The client then passes the unmodified handle back to the server in a subsequent remote call. The called procedure interprets the context handle. For example, to specify a procedure that a client can use to obtain a context handle, you can define the following:

```
typedef [context_handle] void * my_context;
void op1(
  [in]handle_t h,
  [out] my_context * this_object);
```

To specify a procedure that a client can call to make use of a previously obtained context handle, you can define the following:

```
void op2([in] my_context this_object);
```

To close a context, and to clean the context on the client side, you can define the following:

```
[in, out, context_handle] void * my_context;
```

The resources associated with a context handle are reclaimed when, and only when, the manager changes the value of the **in,out** context handle parameter from non-NULL to NULL.

The Context Rundown Procedure

Some uses of context handles may require you to write a context rundown procedure in the application code for the server. If communications between the client and server are broken while the server is maintaining context for the client, RPC invokes the context rundown procedure on the server to recover the resources represented by the context handle. If you declare a context handle as a named type, you must supply a rundown procedure for that type.

When a context requires a context rundown procedure, you must define a named type that has the **context_handle** attribute. For each different context handle type, you must provide a context rundown procedure as part of the manager code.

The format for the rundown procedure name is as follows:

*context_type_name***_rundown**

A rundown procedure takes one parameter, the handle of the context to be run down, and delivers no result. For example, if you declare the following:

```
typedef [context_handle] void * my_context;
```

then the rundown procedure is as follows:

```
void my_context_rundown (my_context this_object);
```

Server application code that uses a certain context handle may be executing in one or more server threads at the time that RPC detects that communications between the server and the client that is using that context have broken. The context rundown routine will not be invoked until a return of control to the server stub has happened in each of the threads that were using the context handle.

If application code in any of these threads destroys the context before returning control to the server stub from which it was called, your context rundown procedure will not be executed.

Creating New Context

When a client makes its first request to the manager to manipulate context, the manager creates context information and returns this information to the client through a parameter of the type **context_handle**. This parameter must be an output parameter or an input/output parameter whose value is NULL when the call is made. A context handle can also be a function result.

Reclaiming Client Memory Resources for the Context Handle

In the event that a communications error causes the context handle to be unusable, the resources that maintain the context handle must be reclaimed. Use the **rpc_ss_destroy_client_context()** routine in the client application to reclaim the client-side resources and to set the context handle value to NULL.

The syntax of the routine is as follows:

```
void rpc_ss_destroy_client_context(
   void *p_unusable_context_handle);
```

Relationship of Context Handles and Binding

For the client, the context handle specifies the state within a server and also contains binding information. If an operation has an input context handle or input/output context handle that is not NULL, it is not necessary to supply any other binding information. A context handle that has only the **in** attribute cannot be NULL. If an operation has **in,out** context handle parameters but no **in** context handle parameters, at least one of the **in,out** context handle parameters cannot be NULL. However, if the only context handle parameters in an operation are output, they carry no binding information. In this case, you must use another method to bind the client to a server.

If you specify multiple context handles in an operation, all active context handles must map to the same remote address space on the same server or the call fails. (A context handle is active while it represents context information that the server maintains for the client. It is inactive if no context has yet been created, or if the context is no longer in use.)

Rules for Using Context Handles

The following rules apply to using context handles:

- A context handle can be a parameter or a function result. You cannot use context handles as an array element, as a structure or union member, or as the element type of a pipe.
- A context handle cannot have the **transmit_as** or **ptr** attributes.
- An input-only context handle cannot be NULL.
- A context handle cannot be pointed to, except by a top-level reference pointer.

Examples Using Context Handles

The following examples show a sample IDL file that uses context handles and a sample context rundown procedure file.

Example of an IDL File That Uses a Context Handle

```
/*
 * Filename: context_handle.idl
 */
[uuid(f38f5080-2d27-11c9-a96d-08002b0ecef1),
 pointer_default(ref), version (1.0)]
interface files
{
/* File context handle type */
typedef [context_handle] void * file_handle_t;
/* File specification type */
typedef [string] char * filespec_t;
/* File read buffer type */
typedef [string] char buf_t[*];

 /*
 * The file_open call requires that the client has located a
 * file server interface files and that an RPC handle that is
 * bound to that server be passed as the binding parameter h.
 *
 * Operation to OPEN a file; returns context handle for that
 * file.
 */
file_handle_t file_open
(
 /* RPC handle bound to file server */
  [in] handle_t h,
 /* File specification of file to open */
  [in] filespec_t fs
);
 /*
 * The file_read call is able to use the context handle
 * obtained from the file_open as the binding parameter,
 * thus an RPC handle is not necessary.
 *
 * Operation to read from an opened file; returns true if
 * not end-of-file
 */
boolean file_read
(
 /* Context handle of opened file */
  [in] file_handle_t fh,
 /* Maximum number of characters to read */
  [in] long buf_size,
 /* Actual number of characters of data read */
  [out] long *data_size,
 /* Buffer for characters read */
  [out, size_is(buf_size), length_is(*data_size)] \
   buf_t buffer
);
 /* Operation to close an opened file */
void file_close
(
 /* Valid file context handle goes [in]. On successful close,
 * null is returned.
 */
  [in,out] file_handle_t *fh
);
}
```

Example of a Context Rundown Procedure

```
/*
 * fh_rundown.c: A context rundown procedure.
 */
```

```
#include <stdio.h>
#include "context_handle.h"  /* IDL-generated header file */

void file_handle_t_rundown
(
  file_handle_t file_handle  /* Active context handle */
                 * (open file handle)  */
)

{
  /*
   * This procedure is called by the RPC runtime on the
   * SERVER side when communication is broken between the
   * client and server. This gives the server the
   * opportunity to reclaim resources identified by the
   * passed context handle. In this case, the passed
   * context handle identifies a file, and simply closing
   * the file cleans up the state maintained by the context
   * handle, that is "runs down" the context handle. Note
   * that the file_close manager operation is not used here;
   * perhaps it could be, but it is more efficient to use
   * the underlying file system call to do the close.
   *
   * File handle is void*, it must be cast to FILE*
   */
  fclose((FILE *)file_handle);
}
```

IDL Support for C++

Most of the IDL features apply to both C and C++ applications. However, just as C++ is an extension to C, this section describes additional IDL features required to use IDL effectively with C++.

When the IDL compiler compiles an interface definition, it typically generates a header file and one or more intermediate stub files in C or C++, and then it invokes the appropriate compiler to generate object stub files. The IDL Compiler generates C language intermediate stub files by default, but you can use the **-lang cxx** option to cause it to generate C++ files instead.

This section describes the following topics:

- The **idl**-generated class hierarchy

  This is important for a basic understanding of how to integrate the interface into an object-oriented application.
- Interface inheritance

  One interface can be derived from another, just as classes are derived from other classes.
- Static operations

  Static operations specify member functions that are called independently from an object. All other operations specify nonstatic member functions which are only invoked with respect to an existing object.
- Reference parameters

  Reference parameters are passed by reference instead of being passed by value.
- **idl**-generated member functions

  Several member functions are generated by IDL and made part of the interface class. These functions perform useful operations for all interfaces.

The idl-generated Class Hierarchy

For C++ applications, the interface definition specifies a public interface class. This means that IDL data types specify public data members of the interface class, and IDL operations specify member functions. The IDL compiler generates this *interface class* within C++ class hierarchies for both the client and server. The RPC network mechanisms are encapsulated in a class above the interface class. Clients use (and servers implement) the objects of classes below the interface class.

The rpc_object_reference Base Class

Because C++ makes it easy to hide information, the IDL compiler generates an **rpc_object_reference** *base class* for identifying, distributing, and tracking objects. All interface classes inherit the **rpc_object_reference** class, which encapsulates the following information:

- Object binding information, including server binding information and an object UUID representing the object on the particular server
- Transport protocol information for the server
- A name identifying an optional location in the namespace for the object's binding information
- A location flag indicating whether the object is on the local system or a remote system
- A reference count to keep track of how many clients currently access the object

The Interface Class

For each interface, the IDL compiler generates and places in the header file the interface class derived from the **rpc_object_reference** class. The class name generated is the interface name specified in the interface definition. For example, the compiler generates the following class:

```
class interface_name : public virtual rpc_object_reference
```

This is an abstract class that contains public functions for all the operations specified in the IDL interface. The member functions that are not static object creator functions are defined as pure virtual functions. In C++, an abstract class contains at least one pure virtual function, which means that the implementation is postponed until a later, derived class. Therefore, object instances cannot be created for abstract classes, and thus the interface class is not implemented but is only a declaration. Other classes must be derived from the interface class so that objects can be created for clients and servers.

No constructor operations are allowed in the interface definition, and the IDL compiler does not generate one because no objects are created for the interface class. No destructor operations are allowed in the interface definition, but the compiler generates one automatically for the interface class.

The Client's Proxy Class

The IDL compiler places in the header file a *proxy class* derived from the interface class. An instance of a proxy class is also known as an *object reference*, which clients use to access a remote object. This class provides proxy (or surrogate) objects on the client whose member functions (or methods) transparently perform the RPCs that invoke the actual remote object's member functions on the server. The proxy class name is generated from the interface name and the word **Proxy**, as follows:

```
class interface_name Proxy : public interface_name
```

Implementations of the proxy class's member functions are automatically generated in the client stub and represent the client's implementation of the interface's operations.

The Server's Manager Class

A *manager class* is required for servers to implement the interface. The class is generated by the IDL compiler and derived from the interface class as follows:

```
class interface_name_Mgr : public  interface_name  {
public:
  .
  .
  .
}
```

The class is placed in a header file whose name is based on the IDL file and an **_mgr** suffix. When generated, the manager class contains empty functions of all the nonstatic member functions of the interface class. The member function implementations and other implementation details of this class are called *manager code*.

The Interface Inheritance Operator

An interface definition can inherit properties of a previously defined interface, just as a C++ class can inherit properties of previously defined classes. You can modify an interface definition to inherit an interface by using the inheritance operator (**:**) in the interface header, as follows:

```
[interface_attribute, ...] interface interface_name [ : inherited_interface ]
```

This **idl**-generated header file contains the inherited interface's data types and interface class. The interface definition must also declare the information in the inherited interface's header file by using an **import** declaration in the body of the interface. The following example shows how the **derived** interface inherits another interface and imports that interface's definition file. The inherited interface definition file is named **inherit.idl**, and the interface it contains is named **inherit**.

```
interface derived : inherit
{
import "inherit.idl";
.
.
.
}
```

A interface may inherit only one interface; that is, multiple interface inheritance is not allowed.

The static Keyword for Operations

In C++ applications, the interface definition operations specify the member functions of the interface class. The majority of the member functions are invoked by an existing object, but some operations are intended to work regardless of whether an object invokes them or not. Static member functions are invoked independently from any object and are good for such things as object creator functions and for obtaining a class's static data; that is, data that is class-wide and independent from a specific object.

Static member functions may be specified in an interface definition by using the **static** keyword in front of the operation, in one of the following ways:

```
static return_type operation_identifier(...);
```

```
[operation_attribute, ...] static return_type operation_identifier(...);
```

Instead of using the **static** keyword in the interface definition, you can use the **cxx_static** attribute in an ACF.

Since non-static member functions are invoked by an object for which the application must already have a binding, nonstatic operations cannot have a binding handle parameter. If you want to use explicit binding for an interface, only static operations can have a binding handle as the first parameter.

The C++ Reference Operator (&) on Parameters

C++ passes arguments by value; however, to override this and cause a member function argument to be passed by reference, apply the reference operator (**&**) to the parameter in the interface definition. Specify a reference parameter as follows:

```
[parameter_attribute,...] parameter_type  &parameter
```

Using the reference operator on a parameter is the same as applying the reference pointer attribute (**[ref]**) to a pointer parameter.

Functions Generated by IDL

The IDL compiler generates some additional member functions for an interface class. For clients, these functions include overloaded static member functions to bind to remote named objects in various ways and a member function to set security information. For servers, additional member functions exist to advertise named objects and get the binding handle on which a member function was called from within the member function implementation.

Four overloaded functions for binding clients to known objects are named **bind()**. The functions otherwise differ by the type of parameter passed in. Three of these functions are intended for use with named objects and one is used to swap between interfaces when an object supports more than one interface. Each **bind()** function obtains an *object reference* (instance of a proxy class) by returning a pointer to the interface class. These functions are described in the following sections.

The bind() Function for Binding by an Object’s Name

A client can bind to a named object (an object whose name is advertised in a name service) by calling the **idl**-generated **bind()** static member function with the name service name as an argument. The function’s prototype is as follows:

```
interface_class * interface_class::bind(unsigned_char_t *)
```

This function takes a pointer argument of type **unsigned_char_t** that points to a name service name. The function returns a pointer to the interface class. For example:

```
char *CDS_name = "/.:/object_name";
IF_class * object = IF_class::bind((unsigned_char_t *) CDS_name);
```

If the entry contains more than one binding, one is returned at random. The function obtains a full binding (the binding information includes a server’s endpoint.)

Before a client uses this function, a persistent object on a server typically places its name and binding information in the name service by using the **idl**-generated **register_named_object()** member function.

The bind() Function for Binding by an Object’s UUID

A client can bind to a named object by using the object’s UUID as an argument to the **idl**-generated **bind()** static member function. This function’s prototype is as follows:

```
interface_class * interface_class::bind(uuid_t &)
```

This function takes an argument of type **uuid_t** that is the UUID of a named object. The function returns a pointer to the interface class. For example:

```
uuid_t objectUUID;
interface_class * object = interface_class::bind(objectUUID);
```

The search in the namespace for an entry that contains the matching UUID begins with the default entry named in the **RPC_DEFAULT_ENTRY** environment variable. The binding obtained is fully bound.

The bind() Function for Binding by Binding Handle

A client can bind to a known object directly by using a binding handle as an argument to the **idl**-generated **bind()** static member function. This function’s prototype is as follows:

```
interface_class * interface_class::bind(rpc_binding_handle_t)
```

This function takes an argument of type **rpc_binding_handle_t** that is a binding handle to an object. The function returns a pointer to the interface class. For example:

```
rpc_binding_handle_t bindingHandle;
interface_class * object = interface_class::bind(bindingHandle);
```

This function does not use the name service because the client obtains the binding information and binding handle prior to the call. The binding handle can be either partially or fully bound. If the binding handle is partially bound, it becomes fully bound when the object calls a member function.

The bind() Function for Binding by Object Reference

Depending on the application, objects can have the behavior of more than one interface class. However, your code can access only one interface's member functions at a time. A client uses the **idl**-generated **bind()** static member function with an existing object reference as an argument to bind to a different interface. This function's prototype is as follows:

```
different_interface_class * different_interface_class::bind(rpc_object_reference *)
```

This function takes a pointer argument of type **rpc_object_reference** that is an existing object reference to an interface class. The function returns a pointer to a different interface class that the object also supports. The original object is obtained through a previous **bind()** call, an object creator function, or an output parameter of a member function. For example:

```
rpc_binding_handle_t bindingHandle;
interface_class * object = interface_class::bind(bindingHandle);
diff_if_class * new_object = diff_if_class::bind(object);
```

The secure() Function for Setting Object Security

Objects use the **idl**-generated **secure()** member function to set their authorization and authentication information from the client. This sets the information for all the binding handles encapsulated in the client proxy object. The **secure()** function is a public member function of the **rpc_object_reference** class. This function's prototype is as follows:

```
void  interface_class::secure(
  [unsigned_char_t *  server_principal_name =0, ]
  [unsigned32  protection_level = rpc_c_protect_level_default, ]
  [unsigned32  authentication_protocol =rpc_c_authn_default, ]
  [rpc_auth_identity_handle_t authorization_identity =NULL, ]
  [unsigned32  authorization_policy  =rpc_c_authz_name ]
)
```

The following code shows an example of how to use the secure() member function:

```
Matrix     *m;

cout << "creating a remote matrix" << endl;
m = Matrix::createMatrix(1, 2, 3, 4);

cout << "calling set() operation without authorization" << endl;
m->set(0,0,99);
// Without authorization, operation should have not changed anything.
assert(m->get(0,0) == 1);

cout << "setting security privileges on object" << endl;
m->secure(
(unsigned_char_t *) "refmon_test", // server principal name
rpc_c_protect_level_pkt_integ, // protection level
rpc_c_authn_dce_secret, // authentication protocol
NULL, // inherited login context
rpc_c_authz_name // authorization policy
);
```

```
// since we now have set security flags, the set() operation
// should work
m->set(0,0,99);
assert(m->get(0,0) == 99);
```

The example shows that unauthorized use of a matrix's member function will not change values (**m->get(0,0,99);**). However, after setting the appropriate authorization and authentication information with **secure()**, the member function will work as expected. All parameters to the **secure()** function are optional, but it is recommended that you specify values rather than depend on the default values.

The SetRebind( ) Function

A client can automatically rebind to an object if the client first sets a rebind policy by using the **SetRebind()** function. This allows a degree of fault tolerance in an application. For example, if a server goes down and is restarted, the client can re-establish communications with the new server. In another example, if a server provides access over multiple protocols or addresses and one of those links fails, the client can choose another link automatically. Finally, if multiple servers support the same object and one server exits, clients can still access the object via another server.

The format of the function is as follows:

```
void interface_class::SetRebind(
    DCERebindPolicy  policy,
     [ unsigned32 *  n = 0 ]
)
```

The second argument is optional and only used when the rebind policy is **attempt_rebind_n**. The valid policies include the following:

attempt_rebind_n

If a communication fails, try to communicate with the object by selecting another binding until successful or until *n* attempts have been tried.

wait_on_rebind

If a communication fails, try to communicate with the object by selecting another binding until successful or until the calling thread is canceled.

attempt_rebind

If a communication fails, try to communicate with the object by selecting another binding, if possible. If all handles have been tried, return an error. This is the default policy.

never_rebind

If a communication fails, return an error.

The register_named_object() Function

Persistent objects can name and register themselves from the server by using the **idl**-generated **register_named_object()** member function. This function performs the following tasks:

- Creates the name service entry (if it doesn't already exist) and adds the server's binding information so clients can find the server's host
- Replaces or adds the object's binding information in this host's endpoint map so clients can find this server

This function's prototype is as follows:

```
void  interface_class::register_named_object(
        unsigned_char_t *  name_service_name
     [, boolean32  replace_endpoint = TRUE ]
)
```

The function takes a pointer argument of type **unsigned_char_t**, representing the name to use for the name service entry. The function has an optional second argument of type **boolean32** to indicate whether to replace or add the object's binding information to the host's endpoint map. If the second argument is not used (or is set to **TRUE**) and the object's binding information already exists, this function replaces the information. If the second argument is set to **FALSE**, the object's binding information is added to the endpoint map (not replaced). You should add the binding information rather than replace it in circumstances where a single host has more than one server that offers the same interface. The function does not return a value.

The get_binding_handle() Function

Server manager code uses the **idl**-generated **get_binding_handle()** function to obtain the binding handle used to invoke the call. The function's prototype is as follows:

```
rpc_binding_handle_t  get_binding_handle();
```

Member functions (that are not static) cannot have an explicit binding handle argument since the handle is encapsulated in the **rpc_object_reference** base class. A member function implementation uses this function to obtain the binding handle to verify security information, among other things.

Associating a Data Type with a Transmitted Type

The **transmit_as** attribute associates a transmitted type that stubs pass over the network with a presented type that clients and servers manipulate. The specified transmitted type must be a named type defined previously in another type declaration.

There are two primary uses for this attribute:

- To pass complex data types for which the IDL compiler cannot generate marshalling and unmarshalling code.
- To pass data more efficiently. An application can provide routines to convert a data type between a sparse representation (presented to the client and server programs) and a compact one (transmitted over the network).

To build an application that uses presented and transmitted types, you must write routines to perform conversions between the types and to manage storage for the types, and you must link those routines with your application code. At runtime, the client and server stubs call these routines before sending and after receiving data of these types.

The following paragraphs specify C prototypes for generic binding and unbinding routines; in these prototypes, *PRES* is the name of the presented type and *TRANS* is the name of the transmitted type.

The *PRES*_**to_xmit()** routine allocates storage for the transmitted type and converts from the presented type to the transmitted type:

```
void PRES_to_xmit (PRES *presented, TRANS **transmitted)
```

The *PRES*_**from_xmit()** routine converts from the transmitted type to the presented type and allocates any storage referenced by pointers in the presented type:

```
void PRES_from_xmit (TRANS *transmitted, PRES *presented)
```

The *PRES*_**free_inst()** routine frees any storage referenced by pointers in the presented type by *PRES*_**from_xmit()**:

```
void PRES_free_inst (PRES *presented)
```

Suppose that the **transmit_as** attribute appears either on the type of a parameter or on a component of a parameter and that the parameter has the **out** or **in,out** attribute. Then, the *PRES*_**free_inst()** routine will be called automatically for the data item that has the **transmit_as** attribute.

Suppose that the **transmit_as** attribute appears on the type of a parameter and that the parameter has only the **in** attribute. Then, the *PRES*_**free_inst()** routine will be called automatically.

Finally, suppose that the **transmit_as** attribute appears on a component of a parameter and that the parameter has only the **in** attribute. Then, the *PRES*_**free_inst()** routine will not be called automatically for the component; the manager application code must release any resources that the component uses, possibly by explicitly calling the *PRES*_**free_inst()** routine.

The *PRES*_**free_xmit()** routine frees any storage that has been allocated for the transmitted type by *PRES*_**to_xmit()**:

```
void PRES_free_xmit (TRANS *transmitted)
```

A type with the **transmit_as** attribute cannot have other type attributes, specifically the following:

- A pipe type.
- A pipe element type.
- A type with the **context_handle** attribute.
- A type of which any instance has the **context_handle** attribute.
- A type that includes the **handle** attribute in its definition cannot be used, directly or indirectly, in the definition of a type with the **transmit_as** attribute. Nor can a type that includes the **transmit_as** attribute in its definition be used, directly or indirectly, in the definition of a type with the **handle** attribute.
- A conformant array type.
- A varying array type.
- A structure type containing a conformant array.
- An array type of which any instance is varying.
- A type with the **represent_as** attribute.

The type name in a declaration for a **transmit_as** attribute is restricted to 21 characters.

A transmitted type specified by the **transmit_as** attribute must be either a base type, a predefined type, or a named type defined via **typedef**. A transmitted type cannot be a conformant array type or a conformant structure type if any instance of that type is an **in** parameter or an **in, out** parameter.

The following is an example of **transmit_as**. Assuming the following declarations:

```
typedef
  struct tree_node_t {
   data_t data;
   struct tree_node_t * left;
   struct tree_node_t * right;
  } tree_node_t;

typedef
  [transmit_as(tree_xmit_t)] tree_node_t *tree_t;
```

The application code must include routines that match the prototypes:

```
void tree_t_to_xmit (tree_t *, (tree_xmit_t **));
void tree_t_from_xmit ((tree_xmit_t *), (tree_t *));
void tree_t_free_inst (tree_t *);
void tree_t_free_xmit ((tree_xmit_t *)
);
```

IDL Grammar Synopsis

This section summarizes IDL syntax, in extended Backus-Naur Format (BNF) notation.

```
<interface> ::= <interface_header> "{" <interface_body> "}"

<interface_header> ::=
  "[" <interface_attributes> "]" "interface" <identifier> \
       [ ":" <identifier> ]

<interface_attributes> ::=
 <interface_attribute> [ "," <interface_attribute> ] ...

<interface_attribute> ::= "uuid" "(" <uuid_rep> ")"
 | "version" "(" <major> [ "." <minor> ] ")"
 | "endpoint" "(" <endpoint_spec> [ "," <endpoint_spec> ] ... ")"
 | "pointer_default" "(" <pointer_attribute> ")"
 | "local"
 | "exceptions" "(" <excep_name> ["," <excep_name>] ... ")"

<excep_name> ::= <Identifier>

<major> ::= <integer>

<minor> ::= <integer>

<endpoint_spec> ::=
 """ <family_string> ":" "[" <endpoint_string> "]" """

<family_string> ::= <identifier>

<endpoint_string> ::= <identifier>

<interface_body> ::= [ <import> ] ... [ <export> ] ...

<export> ::= <const_declaration> ";"
 | <type_declaration> ";"
 | <op_declaration> ";"
```

```
<import> ::= import <import_files> ";"

<import_files> ::= <filename> [ "," <filename> ] ... ";"

<filename> ::= """ <character> ... """

<const_declaration> ::=
 "const" <const_type_spec> <identifier> "=" <const_exp>

<const_type_spec> ::=
 <integer_type> | "char" | "char" "*" | "boolean" | "void" "*"

<const_exp> ::=
 <integer_const_exp> | <character_const> | <string_const>
 | <identifier> | "TRUE" | "FALSE" | "NULL"

<integer_const_exp> ::= <conditional_exp>

<conditional_exp> ::= <logical_or_exp>
 | <logical_or_exp> "?" <integer_const_exp> ":" <conditional_exp>

<logical_or_exp> ::= <logical_and_exp>
 | <logical_or_exp> "||" <logical_and_exp>

<logical_and_exp> ::= <inclusive_or_exp>
 | <logical_and_exp> "&&" <inclusive_or_exp>

<inclusive_or_exp> ::= <exclusive_or_exp>
 | <inclusive_or_exp> "|" <exclusive_or_exp>

<exclusive_or_exp> ::= <and_exp>
 | <and_exp> "^" <and_exp>

<and_exp> ::= <equality_exp>
 | <and_exp> "&" <equality_exp>

<equality_exp> ::= <relational_exp>
 | <equality_exp> "==" <relational_exp>
 | <equality_exp> "!=" <relational_exp>

<relational_exp> ::= <shift_exp>
 | <relational_exp> "<" <shift_exp>
 | <relational_exp> ">" <shift_exp>
 | <relational_exp> "<=" <shift_exp>
 | <relational_exp> ">=" <shift_exp>

<shift_exp> ::= <additive_exp>
 | <shift_exp> "<<" <additive_exp>
 | <shift_exp> ">>" <additive_exp>

<additive_exp> ::= <multiplicative_exp>
 | <additive_exp> "+" <multiplicative_exp>
 | <additive_exp> "-" <multiplicative_exp>

<multiplicative_exp> ::= <unary_exp>
 | <multiplicative_exp> "*" <unary_exp>
 | <multiplicative_exp> "/" <unary_exp>
 | <multiplicative_exp> "%" <unary_exp>
<unary_exp> ::= <primary_exp>
 | "+" <primary_exp>
 | "-" <primary_exp>
 | "~" <primary_exp>
 | "!" <primary_exp>

<primary_exp> ::= <integer_literal>
 | <identifier>
```

```
<character_const> ::= "'" <character> "'"

<string_const> ::= """ [ <character> ] ... """

<type_declaration> ::=
 "typedef" [ <type_attributes> ] <type_spec> <declarators>

<type_spec> ::= <simple_type_spec>
 | <constructed_type_spec>

<simple_type_spec> ::= <base_type_spec>
 | <predefined_type_spec>
 | <identifier>

<declarators> ::= <declarator> [ "," <declarator> ] ...

<declarator> ::= <simple_declarator>
 | <complex_declarator>

<simple_declarator> ::= <identifier>

<complex_declarator> ::= <array_declarator>
 | <function_ptr_declarator>
 | <ptr_declarator>
 | <ref_declarator>

<ref_declarator> ::= "&" <identifier>

<tagged_declarator> ::= <tagged_struct_declarator>
 | <tagged_union_declarator>

<base_type_spec> ::= <integer_type>
 | <floating_type>
 | <char_type>
 | <boolean_type>
 | <byte_type>
 | <void_type>
 | <handle_type>

<floating_type> ::= "float" | "double"

<integer_type> ::= <signed_int> | <unsigned_int>

<signed_int> ::= <int_size> [ "int" ]
<unsigned_int> ::= <int_size> "unsigned" [ "int" ]
 | "unsigned" <int_size> [ "int" ]

<int_size> ::= "hyper" | "long" | "short" | "small"

<char_type> ::= [ "unsigned" ] "char"

<boolean_type> ::= "boolean"

<byte_type> ::= "byte"

<void_type> ::= "void"

<handle_type> ::= "handle_t"

<constructed_type_spec> ::= <struct_type>
 | <union_type>
 | <tagged_declarator>
 | <enumeration_type>
 | <pipe_type>

<tagged_struct_declarator> ::= "struct" <tag>
```

```
 | <tagged_struct>

<struct_type> ::= "struct" "{" <member_list> "}"

<tagged_struct> ::= "struct" <tag> "{" <member_list> "}"

<tag> ::= <identifier>

<member_list> ::= <member> [ <member> ] ...

<member> ::= <field_declarator> ";"

<field_declarator> ::= [ <field_attribute_list> ]
 <type_spec> <declarators>

<field_attribute_list> ::= "[" <field_attribute> [ ","
 <field_attribute>] ... "]"

<tagged_union_declarator> ::= "union" <tag>
 | <tagged_union>

<union_type> ::= "union" <union_switch> "{" <union_body>
 "}" | "union" "{" <union_body_n_e> "}"

<union_switch> ::= "switch" "(" <switch_type_spec> <identifier> ")"
 [ <union_name> ]
<switch_type_spec> ::= <integer_type>
 | <char_type>
 | <boolean_type>
 | <enumeration_type>
<tagged_union_declarator> ::= "union" <tag>
 | <tagged_union>

<union_type> ::= "union" <union_switch> "{" <union_body> "}"
 | "union" "{" <union_body_n_e> "}"

<union_switch> ::= "switch" "(" <switch_type_spec> \
   <Identifier> ")" [ <union_name> ]

<switch_type_spec> ::= <primitive_integer_type>
 | <char_type>
 | <boolean_type>
 | <enumeration_type>

<tagged_union> ::= "union" <tag> <union_switch> "{"
<union_body> "}"
 | "union" <tag> "{" <union_body_n_e> "}"

<union_name> ::= <Identifier>

<union_body> ::= <union_case> [ <union_case> ] ...

<union_body_n_e> ::= <union_case_n_e> [ <union_case_n_e> ] ...

<union_case> ::= <union_case_label> \
   [ <union_case_label> ] ... <union_arm> | <default_case>

<union_case_n_e> ::= <union_case_label_n_e> <union_arm>
 | <default_case_n_e>

<union_case_label> ::= "case" <const_exp> ":"

<union_case_label_n_e> ::= "[" "case" "(" <const_exp> \
   [ , <const_exp>] ...")" "]"

<default_case> ::= "default" ":" <union_arm>
```

```
<default_case_n_e> ::= "[" "default" "]" <union_arm>

<union_arm> ::= [ <field_declarator> ] ";"

<union_type_switch_attr> ::= "switch_type" "(" \
    <switch_type_spec> ")"

<union_instance_switch_attr> ::= "switch_is" "(" <attr_var> ")"

<enumeration_type> ::=

<enum_identifier> ::= <identifier> [ "=" <const_exp> ]

<pipe_type> ::= "pipe" <type_spec> <pipe_declarators>

<array_declarator> ::= <identifier> <array_bounds_list>

<array_bounds_list> ::= <array_bounds_declarator>
 [ <array_bounds_declarator> ] ...

<array_bounds_declarator> ::= "[" [ <array_bound> ] "]"
 | "[" <array_bounds_pair> "]"

<array_bounds_pair> ::= <array_bound> ".." <array_bound>

<array_bound> ::= "*"
 | <integer_literal>
 | <identifier>

<type_attribute> ::= "transmit_as" "(" <xmit_type> ")"
 | "handle"
 | <usage_attribute>
 | <union_type_switch_attr>
 | <ptr_attr>

<usage_attribute> ::= "string"
 | "context_handle"

<xmit_type> ::= <simple_type_spec>

<field_attribute> ::= "first_is" "(" <attr_var_list> ")"
 | "last_is" "(" <attr_var_list> ")"
 | "length_is" "(" <attr_var_list> ")"
 | "max_is" "(" <attr_var_list> ")"
 | "min_is" "(" <attr_var_list> ")"
 | "size_is" "(" <attr_var_list> ")"
 | <usage_attribute>
 | <union_instance_switch_attr>
 | "ignore"
 | <ptr_attr>

<attr_var_list> ::= <attr_var> [ "," <attr_var> ] ...

<attr_var> ::= [ ["*"]<identifier> ]

<ptr_declarator> ::= "*"<identifier>

<ptr_attr> ::= "ref"
 | "unique"
 | "full"

<op_declarator> ::= [ <operation_attributes> ]
 <simple_type_spec> <identifier> <parameter_declarators>
<operation_attributes> ::= "[" <operation_attribute> [ "static" ]
 [ "," <operation_attribute> ] ... "]"
<operation_attribute> ::= "idempotent"
 | "broadcast"
 | "maybe"
 | "reflect_deletions"
 | <usage_attribute>
 | <ptr_attr>

<param_declarators> ::= "(" "void" ")"
 | "(" [ <param_declarator> [ "," <param_declarator> ] ... ] ")"

<param_declarator> ::= <param_attributes> <type_spec> <declarator>

<param_attributes> ::=
 "[" <param_attribute> [ "," <param_attribute> ] ... "]"

<param_attribute> ::= <directional_attribute>
 | <field_attribute>

<directional_attribute> ::= "in" [ "(" "shape" ")" ]
 | "out" [ "(" "shape" ")" ]

<function_ptr_declarator> ::= <simple_type_spec>
 "(" "*"<identifier> ")" <param_declarators>

<predefined_type_spec> ::= "error_status_t"
 | <international_character_type>

<international_character_type> ::= ISO_LATIN_1
 | ISO_MULTI_LINGUAL
 | ISO_UCS

<pipe_declarators> ::= <pipe_declarator> \
     [ "," <pipe_declarator> ] ...

<pipe_declarator> ::= <simple_declarator>
 | <ptr_declarator>
 | <ref_declarator>
```

Chapter 19. Attribute Configuration Language

The Attribute Configuration Language is used for writing an Attribute Configuration File (ACF). Use the attributes in the ACF to modify the interaction between the application code and stubs without affecting the client/server network interaction.

Syntax Notation Conventions

The syntax of the Attribute Configuration Language is similar to the syntax of IDL. For syntax information, see the syntax notation conventions for the IDL.

The use of **[ ]** (brackets) can be either a required part of the syntax or can denote that a string is optional to the syntax. To differentiate this, brackets that are required are shown as [ ] (plain square brackets). Brackets that contain optional strings are shown as *[ ]* (italicized square brackets).

A **|** (vertical bar) denotes a logical OR.

Attribute Configuration File

The ACF changes the way the IDL compiler interprets the interface definition, written in IDL. The IDL file defines a means of interaction between a client and a server. For new server implementations to be compatible across the network with existing servers, the interaction between the client and server must not be modified. If the interaction between an application and a specific stub needs to change, you must provide an ACF when you build this stub.

The ACF affects only the interaction between the generated stub code and the local application code; it has no effect on the interaction between local and remote stubs. Therefore, client and server writers are likely to have different attribute configuration files that they can modify as desired.

Naming the ACF

To name the ACF, replace the extension of the IDL file (**.idl**) with the extension of the ACF (**.acf**). For example, the ACF associated with *my_idl_filename***.idl** is *my_idl_filename***.acf**.

Compiling the ACF

When you issue the **idl** command, naming the IDL file to compile, it searches for a corresponding ACF and compiles it along with the IDL file. The compiler also searches for any ACF (there can be more than one) associated with any imported IDL files. The stubs that the compiler creates contain the appropriate modifications.

ACF Features

The following list contains the ACF attributes and the features associated with the attributes:

- **include** statement: Includes header files in the generated code
- **auto_handle**, **explicit_handle**, **implicit_handle**, **binding_callout**: Controls binding

- **comm_status**, **fault_status**: Indicates parameters to hold status conditions occurring in the call
- **cs_char**, **cs_tag_rtn**, **cs_stag**, **cs_drtag**, **cs_rtag**: Controls the transmission of international (non-PCS) characters
- **code**, **nocode**: Controls which operations of the IDL file are compiled
- **encode**, **decode**: Controls the generation of IDL encoding services stubs to perform encoding or decoding operations
- **extern_exceptions**: Indicates user-defined parameters to hold status conditions occurring in the call
- **represent_as**: Controls conversion between local and network data types
- **enable_allocate**: Forces the initialization of the memory management routines
- **heap**: Specifies objects to be allocated from heap memory
- **cxx_lookup, cxx_delegate, cxx_new, cxx_static**: Specifies C++ features

Structure

The structure of the ACF is as follows:

```
interface_header
{

interface_body
}
```

Follow these structural rules when creating an ACF:

- The basename of the ACF must be the same as the basename of the IDL file although the extensions are different.
- The interface name in the ACF must be the same as the interface name in the corresponding IDL file.
- With a few exceptions, any type, parameter, or operation names in the ACF must be declared in the IDL file, or defined in files included by use of the **include** statement, as the same class of name.
- Except for additional status parameters, any parameter name that occurs within an operation in the ACF must also occur within that operation in the IDL file.

ACF Interface Header

The ACF interface header has the following structure:

```
[[acf_attribute_list]] interface idl_interface_name
```

The *acf_attribute_list* is optional. The interface header attributes can include one or more of the following attributes, entered within brackets. If you use more than one attribute, separate them with commas and include the list within a single pair of brackets. (Note that some of these attributes can be used in the ACF body also. See “ACF Interface Body” on page 427 for more information.)

- **code**
- **nocode**
- **implicit_handle(** *handle_type handle_name***)**
- **auto_handle**
- **explicit_handle**

- **encode**
- **decode**
- **binding_callout(** *routine_name***)**
- **extern_exceptions(** *exception_name*[**,** *exception_name*]*...***)**
- **cs_tag_rtn(** *tag_set_routine***)**
- **cxx_lookup**(*function_name*)
-

  cxx_delegate(*class_name*)

The following example shows how to use more than one attribute in the ACF interface header:

```
[auto_handle, binding_callout(rpc_ss_bind_authn_client)] \
 interface phone_direct
{
}
```

ACF Interface Body

The ACF interface body can contain the elements in the following list. Note that some of the attributes listed here can also be used in the ACF header, as described in "ACF Interface Header" on page 426. If you use more than one attribute, separate them with commas and include the list within a single pair of brackets.

- An **include** statement
- A declared type

  ```
  typedef [[represent_as (local_type_name)] | [heap] |
  [cs_char (local_type_name)]] type_name;
  ```
- An operation

  ```
  [[explicit_handle] | [comm_status] | [fault_status] |
  [code] | [nocode] | [enable_allocate] |
  [cxx_new(manager_class)] | [cxx_static] | [cxx_static(local_function)] |
  [encode] | [decode] | [cs_tag_rtn (tag_set_routine)]
  ] operation_name ([parameter_list]);
  ```

  A *parameter_list* is a list of zero or more parameter names as they appear in the corresponding operation definition of the IDL file. You do not need to use all the parameter names that occur in the IDL operation definition; use only those to which you attach an ACF attribute. If you use more than one parameter name, the names must be separated by commas.
- A parameter within an operation

  ```
  [[comm_status] | [fault_status] | [heap] |
  [cs_stag] | [cs_drtag] | [cs_rtag]] parameter_name
  ```

The include Statement and the C++ Attributes cstub and sstub

The **include** statement specifies any additional header files you want included in the generated stub code. You can specify more than one header file. The **include** statement is placed in the body of the ACF and has the following syntax:

```
include "filename" [,"filename"] ...;

[ [sstub | cstub | sstub, cstub]] include "filename";
```

Do not specify the directory name or file extension when you use the **include** statement. The compiler appends the **.h** extension. If you want to specify the directory name(s), use the **-cc_opt** or **-I** IDL compiler command options.

Use the **include** statement whenever you use the **represent_as, implicit_handle, cs_char, cxx_static, cxx_new, cxx_lookup**, or **cxx_delegate** attributes and the specified type is not defined or imported in the IDL file.

The **sstub** and **cstub** attributes are optional. By default, the IDL compiler places directives only in the **idl**-generated header file when neither the **sstub** nor **cstub** attributes are used. These attributes restrict where **#include** compiler directives are placed in order to include application-specific header files in C++ client and server stub files. In C++ applications, local and remote versions of operations are included together by linking in both the client and server stubs. C++ applications need to control **#include** compiler directives so that the operations for local objects can be easily renamed to avoid name conflicts with the operations for remote objects.

The following table shows to which output file the IDL compiler places the **#include** compiler directive for the application-specific file. Note, that the idl-generated header file is always included automatically in each stub.

ACF Statement	Header File	Client Stub	Server Stub
include ″ *file*″;	**X**		
[cstub] include ″ *file*″;		**X**	
[sstub] include ″ *file*″;			X
[cstub, sstub] include ″ *file*″;		**X**	X

The auto_handle Attribute

This attribute causes the client stub and RPC runtime to manage the binding to the server by using a directory service. Any operation in the interface that has no parameter containing binding information is bound automatically to a server so the client does not have to specify a binding to a server.

When an operation is automatically bound, the client does not have to specify the server on which an operation executes. If you make a call on an operation without explicit binding information in an interface for which you have specified **auto_handle**, and no client/server binding currently exists, the client stub selects an available server and establishes a binding. This binding is used for this call and subsequent calls to all operations in the interface that do not include explicit binding information, while the server is still available.

When a client uses the automatic binding method, DCE must use the name service to obtain binding information. However, the client host must have a starting entry from which to begin the namespace search. If the **RPC_DEFAULT_ENTRY** environment variable is defined on the client host, DCE uses the entry in that variable to obtain binding information. If **RPC_DEFAULT_ENTRY** is not defined, DCE looks for binding information from the host’s name service profile.

Server termination, network failure, or other problems can cause a break in binding. If this occurs during the execution of an automatically bound operation, the client stub issues the call to another server, provided one is available and the operation is idempotent, or it determines that the call did not start to run on the server. Similarly, if a communications or server failure occurs between calls, the client stub binds to another server for the next call, if a server is available.

If the client stub is unable to find a server to run the operation, it reports this by returning the status code **rpc_s_no_more_bindings** in the **comm_status** parameter, or by raising the exception **rpc_x_no_more_bindings** if the operation does not use the **comm_status** attribute for error reporting. Note that, if a binding breaks, the search for another server begins at the directory service entry following the one where the binding broke. This means that, even if a server earlier in the list becomes available, it is not treated as a candidate for binding. After the RPC runtime tries each server in the list, it reinitializes the list of server candidates and tries again. If the second attempt is unsuccessful, the RPC runtime reports the status code **rpc_s_no_more_bindings**. The next call on an operation in the interface starts from the top of the list when looking for a server to bind to.

The **auto_handle** attribute can occur at most once in the ACF.

If an interface uses the **auto_handle** attribute, the presence of a binding handle or context handle parameter in an operation overrides **auto_handle** for that operation.

The **auto_handle**attribute declaration has the following syntax. (See the example at the end of this section.)

[auto_handle] interface *interface_name*

You cannot use **auto_handle** if you use **implicit_handle** or if you use **explicit_handle** in the interface header. You also cannot use **auto_handle** if you use the **encode** or **decode** ACF attributes.

Example Using the auto_handle Attribute

ACF

```
[auto_handle] interface math_1
{
}
```

IDL File

```
[uuid(b3c86900-2d27-11c9-ab09-08002b0ecef1)]
interface math_1
{
/* This operation has no handle parameter,
 * therefore, uses automatic binding.
 */
long add([in] long a,
     [in] long b);

/*
 * This operation has an explicit handle parameter, h,
 * that overrides the [auto_handle] ACF attribute.
 * Explicit handles also override [implicit_handle].
 */
long subtract ([in] handle_t h,
        [in] long a,
        [in] long b);
}
```

The explicit_handle Attribute

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

The **explicit_handle** attribute has the following syntax. (See the example at the end of this section.)

For an interface:

[explicit_handle] interface *interface_name*

For an operation:

[**explicit_handle]** *operation_name* (*[parameter_list]*);

When used as an ACF interface attribute, the **explicit_handle** attribute applies to all operations in the IDL file. When used as an ACF operation attribute, this attribute applies to only the operation you specify.

If you use the **explicit_handle** attribute as an ACF interface attribute, you must not use the **auto_handle** or **implicit_handle** attributes. Also, you cannot use the **encode** and **decode** attributes if you use **explicit_handle**.

Using the **explicit_handle** attribute on an interface or operation has no effect on operations in IDL that have explicit binding information in their parameter lists.

Example Using the explicit_handle Attribute

ACF

```
[explicit_handle] interface math_2
{

 /* This causes the operation, as called by the client, to
  * have the parameter handle_t IDL_handle, at the start of
  * the parameter list, before the parameters specified here
  * in the IDL file.
  */
}
```

IDL File

```
[uuid(41ce5b80-0ba7-11ca-87ba-08002b111685)]
interface math_2
{
long add([in] long a,
     [in] long b);
}
```

The implicit_handle Attribute

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

The **implicit_handle** attribute declaration has the following syntax. (See the example at the end of this section.)

For an interface:

[**implicit_handle** (*handle_type handle_name*)] **interface** *interface_name*

If an interface uses the **implicit_handle** attribute, the presence of a binding handle or **in** or **in,out** context handle parameter in an operation overrides the implicit handle for that operation.

The **implicit_handle** attribute can occur at most once in the ACF.

You cannot use the **implicit_handle** attribute if you are using the **auto_handle** attribute or the **explicit_handle** attribute as an interface attribute. You also cannot use **implicit_handle** if you use the **encode** or **decode** ACF attributes.

If the type in the **implicit_handle** clause is not **handle_t**, then it is treated as if it has the **handle** attribute.

The ACF in the following example modifies the **math_3** interface to use an implicit handle.

Example Using the implicit_handle Attribute

ACF

```
[implicit_handle(user_handle_t global_handle)] interface math_3
{
/*
 * Since user_handle_t is not a type defined in IDL, you
 * must specify an header file that contains the definition
 */
include "user_handle_t_def";
}
```

IDL File

```
[uuid(a01d0280-2d27-11c9-9fd3-08002b0ecef1)]
interface math_3
{
long add([in] long a,
     [in] long b);
}
```

The client_memory Attribute

While marshalling parameters, the client stub uses built-in routines to manage memory. You can use the **client_memory** attribute to specify different memory allocation and free routines. The **client_memory** attribute has the following syntax in the ACF header:

```
[client_memory(malloc_routine, free_routine)] interface
idl _interface_name
```

The routines you specify must have the same respective procedure declarations as the system's **malloc()** and **free()** routines.

Applications need to manage memory consistently, so if your application needs to do other memory allocation, be sure to use the same routines you specified with the **client_memory** attribute.

You can use the **client_memory** attribute in conjunction with RPC stub support API routines such as **rpc_sm_set_client_alloc_free()** and **rpc_sm_swap_client_alloc_free()**.

The comm_status and fault_status Attributes

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

The **comm_status** attribute causes communications failures to be reported through a specified parameter. The **fault_status** attribute causes server failures to be reported through a specified parameter. Applying both attributes causes all remote and communications failures to be reported through status. Any local exception caused by an error during marshalling, correctness checking performed by the client stubs, or an error in application routines continues to be returned as an exception.

The **comm_status** and **fault_status** attributes have the following syntax. (See the examples at the end of this section.)

For an operation:

```
[comm_status | fault_status] operation_name ([parameter_list]);
```

For a parameter:

```
operation_name ([comm_status | fault_status] parameter_name);
```

Note: You can apply one of each attribute to the same operation and/or parameter at the same time. Separate the attributes with a comma. (See the example at the end of this section.)

If the parameter named in a **comm_status** or **fault_status** attribute is in the parameter list for the operation in the IDL file, then it must have the **out** attribute in the IDL file. (Additional ACF parameters do not have **in** and **out** directional attributes.)

If the status attribute occurs on the operation, the returned value result must be defined as type **error_status_t** in the IDL file. If an error occurs during execution of the operation, the error code is returned as the operation result. If the operation completes successfully, the value returned to the client is the value returned by the manager code.

Note: The **error_status_t** type is equivalent to **unsigned32**, which is the data type used by the RPC runtime for an error status. The status code **error_status_ok** is equivalent to **rpc_s_ok**, which is the RPC runtime success status code.

If the status attribute occurs on a parameter, the parameter name does not have to be defined in the IDL file, although it can be. Note the following:

- If the parameter name is one used in the IDL file, then that parameter must be an output parameter of type **error_status_t**. If the operation completes successfully, the value of this parameter is the value returned by the manager code.
- If the parameter name is different from any name defined within the operation definition in the IDL file, the IDL compiler creates an extra output parameter of type **error_status_t** in the application code after the last parameter defined in the IDL file. In a successfully completed remote call, this extra parameter has the value **error_status_ok**.

In either case, if an error occurs during the remote call, the error code is returned to the parameter that has the status attribute. (See the *OSF DCE Problem Determination Guide* for an explanation of status codes.)

If you define both additional **comm_status** and additional **fault_status** parameters, they are automatically added at the end of the procedure declaration in the order of specification in the ACF.

In the following example, there are three possible uses of the status attributes: as the operation result of **add**, as a parameter of **subtract** as defined in the IDL file, and as an additional parameter of **multiply**.

Example Using the comm_status and fault_status Attributes

ACF

```
[auto_handle] interface math_4
{
[comm_status,fault_status] add();

subtract ([comm_status,fault_status] s);

/*
 * 'sts' is not a parameter in the interface definition of
 * operation 'multiply'. This specifies that the application
 * wants a trailing parameter 'sts' that is of type
 * error_status_t, after the parameters a and b.
 */
multiply ([comm_status] c_sts,[fault_status] f_sts);
}
```

IDL File

```
[uuid(91365000-2d28-11c9-ad5a-08002b0ecef1)]
interface math_4
```

```
{
error_status_t add ([in] double a,
          [in] double b,
          [out] double *c);
double subtract ([in] double a,
        [in] double b,
        [out] error_status_t *s);
double multiply ([in] double a,
         [in] double b);
}
```

server.c

```
/*
 * The three server procedures below illustrate the different
 * models of comm_status and fault_status appearing in the
 * idl and acf declarations above.
 *
 * RPC automatically passes back DCE error codes through
 * comm_status and fault_status. These examples differ in
 * their handling of the nonerror case.
 */

error_status_t add (double a,
          double b,
          double * c)
{
  ...
  *c = answer;

  /*
   * comm_status and fault_status are operation attributes.
   * If no error occurs, the client will see the value that
   * the server returns.
   *
   * We return error_status_ok here for the normal
   * successful case.
   */

  return error_status_ok;
}

double subtract (double a,
         double b,
         error_status_t * s)
{
  /*
   * "s" appears in both the idl definition and the acf
   * specification.
   *
   * In the successful case, the client is returned the
   * value that the server puts in *s. Therefore, assume
   * success here.
   */

  *s = error_status_ok;

  ...
  return answer;
}
double multiply (double a,
         double b,
         error_status_t * c_sts,
     error_status_t * f_sts)
{
  /*
```

```
   * c_sts and f_sts appear in the acf, but do not appear
   * in the idl definition. In this case, c_sts and f_sts
   * are placed at the end of the parameter list generated
   * by the idl compiler. To conform to the prototype
   * generated by idl, your server code must also declare
   * these parameters.
   *
   * In the successful case, c_sts and f_sts are
   * automatically returned to the client as
   * error_status_ok. Even though c_sts and f_sts are
   * parameters to the function, the server code must not
   * modify these parameters or store through them.
   */

  ...
  return answer;
}
```

The code and nocode Attributes

The **code** and **nocode** attributes allow you to control which operations in the IDL file have client stub code generated for them by the compiler. These attributes affect only the generation of a client stub; they have no effect when generating the server stub.

The **code** and **nocode** attributes have the following syntax. (See the example at the end of this section.)

For an interface:

[code | nocode] interface *interface_name*

For an operation:

[code | nocode] *operation_name* (*[parameter_list]*);

When you specify **nocode** as an attribute on an ACF interface, stub code is not generated for the operations in the corresponding IDL interface unless you also specify **code** for the particular operation(s) for which you want stub code generated. Similarly, when you specify **code** (the default) as an attribute on an ACF interface, stub code is generated for the operations in the corresponding IDL interface unless you also specify **nocode** for the particular operations for which you do not want stub code generated.

Do *not* use **nocode** on any of the operations if the compiler is generating only server stub code because it has no effect. Server stubs must always contain generated code for all operations.

In the following example, the IDL compiler generates client stub code for the operations **open**, **read**, and **close**, but not for the operation **write**. An alternative method for specifying the same behavior is to use **[nocode] write()** in the ACF.

Example Using the code and nocode Attributes

ACF

```
[nocode,auto_handle] interface open_read_close
{
```

```
[code] open();
[code] read();
[code] close();
}
```

IDL File

```
[uuid(2166d580-0c69-11ca-811d-08002b111685)]
interface open_read_close
{
void open (...);
void read (...);
void write (...);
void close (...);
}
```

The represent_as Attribute

This attribute associates a local data type that your application code uses with a data type defined in the IDL file. Use of the **represent_as** attribute means that, during marshalling and unmarshalling, conversions occur between the data type used by the application code and the data type specified in the IDL.

The **represent_as** attribute has the following syntax. (See the example at the end of this section.)

typedef [**represent_as** (*local_type_name*)] *net_type_name*;

The *local_type_name* is the local data type that the application code uses. You can define it in the IDL file or in an application header file. If you do not define it in the IDL file, use the **include** statement in the ACF to make its definition available to the stubs.

The *net_type_name* is the data type that is defined in the IDL file.

The **represent_as** attribute can appear at most once in a **typedef** declaration in an ACF.

If you use the **represent_as** attribute, you must write routines that perform the conversions between the local and network types, and routines that release the memory storage used to hold the converted data. The conversion routines are part of your application code.

The suffix for the routine names, the function of each, and where they are used (client or server) appear in the following list:
- **_from_local()**: Allocates storage instance of the network type and converts from the local type to the network type (used for client and server).
- **_to_local()**: Converts from the network type to the local type (used for client and server).
- **_free_inst()**: Frees storage instance used for the network type (used by client and server).
- **_free_local()**: Frees storage used by the server for the local type (used in server). This routine frees any object pointed to by its argument but does not attempt to free the argument itself.

Suppose that the **represent_as** attribute is applied to either the type of a parameter or to a component of a parameter and that the parameter has the **out** or **in,out** attribute. Then, the **_free_local()** routine will be called automatically for the data item that has the type to which the **represent_as** attribute was applied.

Suppose that the **represent_as** attribute is applied to the type of a parameter and that the parameter has only the **in** attribute. Then, the **_free_local()** routine will be called automatically.

Finally, suppose that the **represent_as** attribute is applied to the type of a component of a parameter and that the parameter has only the **in** attribute. Then, the **_free_local()** routine will not be called automatically for the component; the manager application code must release any resources that the component uses, possibly by explicitly calling the **_free_local()** routine.

Append the suffix of the routine name to the *net_type_name*. The syntax for these routines is as follows:

void *net_type_name***_from_local** ((*local_type_name* *), (*net_type_name* **))

void *net_type_name***_to_local** ((*net_type_name* *), (*local_type_name* *))

void *net_type_name***_free_inst** ((*net_type_name* *))

void *net_type_name***_free_local** ((*local_type_name* *))

Example Using the represent_as Attribute

ACF

```
[auto_handle] interface phonedir
{
/*
 * You must specify an included file that contains the
 * definition of my_dir_t.
 */
include "user_types";

/*
 * The application code wants to pass data type my_dir_t
 * rather than dir_t. The [represent_as] clause allows
 * this, and you must supply routines to convert dir_t
 * to/from my_dir_t.
 */
typedef [represent_as(my_dir_t)] dir_t;
}
```

IDL File

```
[uuid(06a12100-2d26-11c9-aa24-08002b0ecef1)]
interface phonedir
{
typedef struct
  {
  short int area_code;
  long int  phone_num;
  char    last_name[20];
  char    first_name[15];
  char    city[20];
  } dir_t;
void add ([in] dir_t *info);
void lookup ([in] char city[20],
       [in] char last_name[20],
       [in] char first_name[15],
       [out] dir_t *info);
void delete ([in] dir_t *info);
}
```

The enable_allocate Attribute

The **enable_allocate** attribute on an operation causes the server stub to initialize the **rpc_ss_allocate()** routine. The **rpc_ss_allocate()** routine requires initialization of its environment before it can be called. The server stub automatically initializes (enables) **rpc_ss_allocate()** if the operation uses either full pointers or a type with the **represent_as** attribute. If the operation does not meet either of these conditions, but the manager application code needs to make use of the **rpc_ss_allocate()** and **rpc_ss_free()** routines, then use the **enable_allocate** attribute to force the stub code to enable.

The **enable_allocate** attribute has the following syntax.

For an operation:

[enable_allocate] *operation_name* (*[parameter_list]*);

Example Using the enable_allocate Attribute

ACF

```
[auto_handle] interface phonedir
{
[enable_allocate] lookup ();
}
```

IDL File

```
[uuid(06a12100-2d26-11c9-aa24-08002b0ecef1)]
interface phonedir
{
typedef struct
  {
  short int area_code;
  long int  phone_num;
  char    last_name[20];
  char    first_name[15];
  char    city[20];
  } dir_t;
void add ([in] dir_t *info);
void lookup ([in] char city[20],
       [in] char last_name[20],
       [in] char first_name[15],
       [out] dir_t *info);
void delete ([in] dir_t *info);
}
```

The heap Attribute

This attribute specifies that the server stub’s copy of a parameter or of all parameters of a specified type is allocated in heap memory rather than on the stack.

The **heap** attribute has the following syntax. (See the example at the end of this section.)

For a type:

```
typedef [heap] type_name;
```

For a parameter:

```
operation_name ([heap] parameter_name);
```

Any identifier occurring as a parameter name within an operation declaration in the ACF must also be a parameter name within the corresponding operation declaration in IDL.

The **heap** attribute is ignored for pipes, context handles, and scalars.

Example Using the heap Attribute

ACF

```
[auto_handle] interface galaxies
{
typedef [heap] big_array;
}
```

IDL File

```
[uuid(e61de280-0d0b-11ca-6145-08002b111685)]
interface galaxies
{
typedef long big_array[1000];
}
```

The extern_exceptions Attribute

By default, the IDL compiler declares and initializes all exceptions listed in an **exceptions** interface attribute in the stub code that it generates. You can use the **extern_exceptions** attribute to override this behavior; the **extern_exceptions** attribute allows you to specify one or more exceptions listed in the exceptions interface attribute that you do not want the idl-generated stub code to declare. If the **extern_exceptions** attribute appears with no list, it has the same effect as if all IDL-defined exceptions were specified in the list.

The **extern_exceptions** attribute has the following syntax. (See the example at the end of this section.)

```
[extern_exceptions (exception_name [,exception_name]...)]
interface  interface_name
```

The **extern_exceptions** attribute indicates that the specified exceptions are defined and initialized in some other external manner before calling the **extern_exceptions** attribute. They may be predefined exceptions (such as **exc_e_exquota**) that were provided by another interface, or exceptions that are defined and initialized explicitly by the application itself.

Example Using the extern_exceptions Attribute

In the following example, the exception named in the list in the **extern_exceptions** attribute in the ACF is not defined or initialized in the idl-generated stub code. All of the other exceptions listed in the **exceptions** interface attribute are defined and initialized in the generated stub.

ACF

```
[extern_exceptions(exc_e_exquota)] interface binop {}
/*
 *The exc_e_exquota exception is a predefined exception
 *(provided in exc_handling.h) and so does not need
 *to be declared and initialized in the idl-generated stub.
 */
```

IDL File

```
[uuid(06255501-08af-11cb-8c4f-08002b13d56d),
version (1.1),
 exceptions (
   exc_e_exquota,
   binop_e_aborted,
   binop_e_too_busy,
   binop_e_shutdown)
] interface binop
 {
   long binop_add(
      [in] long a,
      [in] long b
      );
  }
```

The encode and decode Attributes

The **encode** and **decode** attributes are used in conjunction with IDL encoding services routines (**idl_es** *) to enable RPC applications to encode data types in input parameters into a byte stream and decode data types in output parameters from a byte stream without invoking the RPC runtime. Encoding and decoding operations are analogous to marshalling and unmarshalling, except that the data is stored locally and is not transmitted over the network.

The stubs that perform encoding or decoding operations are different from the stubs that perform RPC operations. The ACF attributes **encode** and **decode** direct the IDL compiler to generate encoding or decoding stubs for operations defined in a corresponding IDL interface rather than generating RPC stubs for those operations.

The **encode** and **decode** attributes have the following syntax. (See the example at the end of this section.)

For an interface:

[encode] | **[decode]** | **[encode,decode]** **interface** *interface_name*

For an operation:

[encode] | **[decode]** | **[encode,decode]** *operation_name* (*[parameter_list]*);

When used as an ACF interface attribute, the **encode** and **decode** attributes apply to all operations defined in the corresponding IDL file. When used as an ACF operation attribute, **encode** and **decode** apply only to the operation you specify. If you apply the **encode** or **decode** attribute to an ACF interface or operation, you must not use the **auto_handle** or the **implicit_handle** ACF attributes.

When you apply the **encode** or **decode** attribute to an operation, the IDL compiler generates IDL encoding services stubs that support encoding or decoding, depending on the attribute used, in the client stub code; it does not generate stub code for the operation in the server stub. To generate an IDL encoding services stub that supports both encoding and decoding, apply both attributes to the operation.

If you apply the **encode** or **decode** attribute to all of the operations in an interface, no server stub is generated. If you apply the **encode** and **decode** attributes to some, but not all, of the operations in an interface, the stubs for the operations that do not have the **encode** and **decode** attributes applied to them are generated as RPC stubs into the server stub module.

When data encoding takes place, only the operation's **in** parameters provide data for the encoding. When data decoding takes place, the decoded data is delivered only to the operation's **out** parameters.

If data is being both encoded and decoded, you generally declare all of the operation's parameters to be **in**,**out**. However, you can encode data by using the **in** parameters of one operation, and decode it by using the **out** parameters of another operation if the types and order of the **in** and **out** parameters are the same. For equivalence, the IDL encoding services treat a function result as an **out** parameter that appears after all other **out** parameters.

In the following example, the IDL compiler generates IDL encoding services stub code for the **in_array_op1**, **out_array_op1**, and **array_op2** operations, but not for the **array_op3** operation. The stub code generated for the **in_array_op1** operation supports encoding, the stub code generated for the **out_array_op1** operation supports decoding, and the stub code generated for the **array_op2** operation supports both encoding and decoding. The stub code generated for the **array_op3** is an RPC client stub. For further information on using the IDL encoding services, see "Chapter 18. Interface Definition Language" on page 357 of this guide and the reference pages for the **idl_es_ *(3rpc)** routines.

Example Using the encode and decode Attributes

ACF

```
interface es_array
{
  [encode] in_array_op1();
  [decode] out_array_op1();
  [encode, decode] array_op2();
}
```

IDL File

```
[uuid(20aac780-5398-11c9-b996-08002b13d56d), version(0)]
interface es_array
{
  void in_array_op1([in] handle_t h, [in] long arr[100]);
  void out_array_op1([in] handle_t h, [out] long arr[100]);
  void array_op2([in] handle_t h, [in,out] long big[100]);
  void array_op3([in] handle_t h, [in,out] long big[100]);
}
```

The cs_char Attribute

The **cs_char** attribute is intended for use in internationalized RPC applications. It is used in conjunction with the **cs_stag**, **cs_drtag**, **cs_rtag** and **cs_tag_rtn** attributes and the DCE RPC routines for automatic code set conversion to provide RPC applications with a mechanism for ensuring character and code set interoperability between clients and servers transferring international (non-PCS) characters.

The **cs_char** attribute is very similar in function to the **represent_as** attribute, in that it associates a local data type that your application code uses with a data type defined in the IDL file. The **cs_char** attribute permits the application code to use the local data type for international character data and converts between the local data type and the format specified in the IDL file when transferring international characters over the network. The **cs_char** ACF attribute permits the conversion of characters, arrays of characters, and strings of characters between the format in which the application code requires them and the format in which they are transmitted over the network.

As with **represent_as**, use of the **cs_char** attribute means that, during marshalling and unmarshalling, conversions occur between the data type that the application code is using and the data type specified in IDL. In the case of **cs_char**, the local data type is automatically converted between the local data type in the local code set encoding and the **idl_byte** data type in the network code set encoding. The network code set is the code set encoding that the application code, through the use of the DCE RPC automatic code set conversion routines, has selected to use when transmitting the international characters over the network.

The **cs_char** attribute differs from the **[transmit_as]** attribute in that it does not affect the network contract between the client and server. It differs from the **[represent_as]** attribute in that multiple data items (for example, the characters of an array or string) can be converted with a single stub call to user-written conversion code, and that the conversion can modify array size and data limit information between what is transmitted over the network and what is used by application code.

The **cs_char** attribute has the following syntax. (See the examples at the end of this section.)

```
typedef [cs_char (local_type_name)] net_type_name;
```

The *local_type_name* is the local data type that the application code uses. You can define it in the IDL file or in an application header file. If you do not define it in the IDL file, use the **include** statement in the ACF to make its definition available to the stubs.

The *net_type_name* is the data type that is defined in the IDL file. When used with the **cs_char** attribute, this data type is always **byte** in the IDL file.

If you use the **cs_char** attribute, you must write the following stub support routines for each local type that you define:

- Routines that check the buffer storage requirements for international character data to be converted to determine whether or not more buffer space needs to be allocated to hold the converted data
- Routines to perform conversion between local and network code sets

The suffix for the routine names, the function of each, and where they are used (client or server) appear in the following list:

- *local_type_name*_**net_size()**: Calculates the necessary buffer size for code set conversion from a local code set to a network code set. Client and server stubs call this routine before they marshall any international character data.
- *local_type_name*_**local_size()**: Calculates the necessary buffer size for code set conversion from a network code set to a local code set. Client and server stubs call this routine before they unmarshall any international character data.
- *local_type_name*_**to_netcs()**: Converts international character data from a local code set to a network code set. Client and server stubs call this routine before they marshall any international character data.
- *local_type_name*_**from_netcs()**: Converts international character data from a network code set to a local code set. Client and server stubs call this routine before they unmarshall any international character data.

You specify the name for the local data type in the *local_type_name* portion of the function name. The name that you specify cannot exceed 20 characters because the entire generated name must not exceed the 31-character limit for C identifiers.

For each piece of international character data being marshalled, the **_net_size** and **_to_netcs** routines are called once each. For each piece of international character data being unmarshalled, the **_local_size** and **_from_netcs** routines are called once each.

DCE RPC provides buffer sizing and code set conversion routines for the **cs_byte** and **wchar_t**data types (the **cs_byte** type is equivalent to the **byte** type). If they meet the needs of your application, you can use these RPC routines (**cs_byte_** *** and **wchar_t_** ***) instead of providing your own routines.

If you do provide your own routines for buffer sizing and code set conversion, they must follow a specific signature. See the reference pages for the **cs_byte_** *****(3rpc)** and **wchar_t_** *****(3rpc)** routines for a complete description of the required signatures for these routines.

When international character data is to be unmarshalled, a stub needs to have received a description of the code set being used before it receives the data. For this reason, the **cs_char** attribute cannot be applied to the base type of a pipe or to a type used in constructing the base type of a pipe.

The **cs_char** attribute also cannot be applied to a type if there is an array that has this type as a base type and the array has more than one dimension, or if the attributes **min_is**, **max_is**, **first_is**, **last_is**, or **string** have been applied to the array. As a result, all instances of the type to which **cs_char** has been applied must be scalars or one-dimensional arrays. Only the **length_is** and/or **size_is** attributes can be applied to these arrays.

The following restrictions apply to the use of variables that appear in array attributes:

- Any parameter that is referenced by a **size_is** or **length_is** attribute of an array parameter whose base type has the **cs_char** attribute cannot be referenced by any attribute of an array parameter whose base type does not have the **cs_char** attribute.

- Any structure field that is referenced by a **size_is** or **length_is** attribute of an array field whose base type has the **cs_char** attribute cannot be referenced by any attribute of an array field whose base type does not have the **cs_char** attribute.

The **cs_char** attribute cannot interact with the **transmit_as** or **represent_as** attributes. This restriction imposes the following rules:

- The **cs_char** attribute cannot be applied to a type that has the **transmit_as** attribute, nor can it be applied to a type in whose definition a type with the **transmit_as** attribute is used.
- The **cs_char** attribute cannot be applied to a type that has the **represent_as** attribute, nor can it be applied to a type in whose definition a type with the **represent_as** attribute is used.
- The **cs_char** attribute cannot be applied to the transmitted type specified in a **transmit_as** attribute or to any type used in defining such a transmitted type.

The **cs_char** attribute cannot be applied to any type that is the type of the referent of a pointer that has a **max_is** or **size_is** attribute applied to it. It also cannot be applied to the base type of an array parameter that has the **unique** or **ptr** attribute applied to it.

An application that uses the **cs_char** ACF attribute cannot use the IDL encoding services **encode** and **decode** ACF attributes.

Examples Using the cs_char Attribute

Arrays of **cs_char** can be fixed, varying, conformant, or conformant varying. The treatment of a scalar **cs_char** is similar to that of a fixed array of one element. The following examples show the relationship between IDL declarations and declarations in the generated header file when the **cs_char** attribute has been applied. The examples assume that the ACF contains the type definition:

```
typedef [cs_char(ltype)] my_byte;
```

For a fixed array, if the IDL file contains

```
typedef struct {
  my_byte fixed_array[80];
} fixed_struct;
```

the declaration generated in the header file is

```
typedef struct {
  ltype fixed_array[80];
} fixed_struct;
```

The number of array elements in the local and network representations of the data must be the same as the array size stated in the IDL.

For a varying array, if the IDL file contains

```
typedef struct {
  long l;
  [length_is(l)] my_byte varying_array[80];
} varying_struct;
```

the declaration generated in the header file is

```
typedef struct {
  idl_long_int l;
  ltype varying_array[80];
} varying_struct;
```

Neither the number of array elements in the local representation nor the number of array elements in the network representation may exceed the array size in the IDL.

For a conformant array, if the IDL file contains

```
typedef struct {
  long s;
  [size_is(s)] my_byte conf_array[];
} conf_struct;
```

the declaration generated in the header file is

```
typedef struct {
  idl_long_int s;
  ltype conf_array[1];
} conf_struct;
```

The number of array elements in the local representation and the number of array elements in the network representation need not be the same. The conversions between these numbers are done in the user-provided **_net_size** and **_local_size** routines.

For a conformant varying array, if the IDL file contains

```
typedef struct {
  long s;
  long l;
  [size_is(s), length_is(l)] my_byte open_array[];
} open_struct;
```

the declaration generated in the header file is

```
typedef struct {
  idl_long_int s;
  idl_long_int l;
  ltype open_array[1];
} open_struct;
```

The maximum number of array elements in the local representation and the maximum number of array elements in the network representation need not be the same. The conversions between these numbers are done in the user-provided **_net_size** and **_local_size** routines.

For fixed or varying arrays, the size of the storage available to hold the local data is determined by the array size specified in IDL and the local type specified in the **cs_char** attribute. For conformant or conformant varying arrays, you must determine the transformations between local storage size and network storage size without reference to the characters being transmitted or received. Where a variable-width character set is in use, this means making the most conservative assumption about the size of the data.

The cs_stag, cs_drtag, and cs_rtag Attributes

The **cs_stag**, **cs_drtag**, and **cs_rtag** attributes are used in conjunction with the **cs_char** and (optionally) the **cs_tag_rtn** attributes and DCE RPC routines for automatic code set conversion to provide internationalized RPC applications with a mechanism to ensure character and code set interoperability between clients and servers handling international character data.

The **cs_stag**, **cs_drtag**, and **cs_rtag** attributes are parameter ACF attributes that correspond to the sending tag, desired receiving tag, and receiving tag parameters defined in operations in the IDL file that handle international character data. These operation parameters tag international characters being passed in the operation's input and output parameters with code set identifying information. The **cs_stag**, **cs_drtag**, and **cs_rtag** ACF parameter attributes declare the tag parameters in the corresponding operation definition to be special code set parameters.

The **cs_stag** attribute has the following syntax:

```
operation_name ([cs_stag] parameter_name);
```

The **cs_stag** attribute identifies the code set used when the client sends international characters to the server. Operations defined in the IDL file that specify international characters in **in** parameters must use the **cs_stag** attribute in the associated ACF.

The **cs_drtag** attribute has the following syntax:

```
operation_name ([cs_drtag] parameter_name);
```

The **cs_drtag** attribute identifies the code set the client would like the server to use when returning international characters.

The **cs_rtag** attribute has the following syntax:

```
operation_name ([cs_rtag] parameter_name);
```

The **cs_rtag** attribute identifies the code set that is actually used when the server sends international characters to the client. Operations defined in the IDL file that specify international characters in **out** parameters must apply the **cs_rtag** attribute in the associated ACF.

Example Using the cs_stag, cs_drtag, and cs_rtag Attributes

Here is an example ACF for an IDL file in which the operation **my_op** has the tag parameters **my_stag**, **my_drtag**, and **my_rtag**, whose types are either **unsigned long** or **[ref] unsigned long**.

```
my_op([cs_stag] my_stag, [cs_drtag] my_drtag,[cs_rtag] my_rtag);
```

For more information about the **cs_stag**, **cs_drtag**, and **cs_rtag** ACF attributes and their use in internationalized RPC applications, see "Chapter 16. Writing Internationalized RPC Applications" on page 281 of this guide.

The cs_tag_rtn Attribute

The **cs_tag_rtn** attribute is an ACF attribute for use in RPC applications that handle international character data. This attribute specifies the name of a user-written routine that the client and server stubs will call to set an operation's code set tag parameters to specific code set values. The **cs_tag_rtn** attribute is an optional ACF attribute that you can use to provide code set tag transparency for callers of your interface's operations. If an operation that transfers international character data has the **cs_tag_rtn** attribute applied to it in the corresponding ACF, the code set tag parameters will not appear in the operation's definition within the generated header file. If the **cs_tag_rtn** attribute is not used, the operation's caller must provide appropriate values to the operation's code set tag parameters before international character data is marshalled.

The **cs_tag_rtn** attribute has the following syntax. (See the example at the end of this section.)

For an interface:

```
[cs_tag_rtn(
tag_set_routine)] interface interface_name
```

For an operation:

```
[cs_tag_rtn(tag_set_routine)] operation_name ([parameter_list]);
```

When used as an ACF interface attribute, the **cs_tag_rtn** attribute applies to all operations defined in the corresponding IDL file. When used as an ACF operation attribute, the **cs_tag_rtn** attribute applies only to the operation you specify.

The *tag_set_routine* is the name of the stub support routine that the client and server stubs will call to set the operation's code set tag parameters. The IDL compiler will generate a function prototype for *tag_set_routine* in the generated header file.

Applications can specify the DCE RPC tag-setting routine **rpc_cs_get_tags()** if it meets their applications' needs, or they can write their own tag-setting routines. The routine name must be distinct from any type name, procedure name, constant name, or enumeration name appearing in the interface definition. It must also have a specific calling signature. See the **rpc_cs_get_tags(3rpc)** reference page for a complete description of the required routine signature.

When the tag-setting routine is called from a client stub, it is called before any **in** parameters are marshalled. When called from a server stub, it is called before any **out** parameters are marshalled. For more information on the **cs_tag_rtn** attribute and its use in internationalized RPC applications, see "Chapter 16. Writing Internationalized RPC Applications" on page 281 of this guide.

Example Using the cs_tag_rtn Attribute

As shown in the following example, the **cs_tag_rtn** attribute is used in conjunction with the **cs_char**, **cs_stag**, **cs_drtag**, and **cs_rtag** ACF attributes. In the example, the stub generated for **a_op** will call the tag-setting routine **set_tags** to set the code set tag parameters to specific values before any data is marshalled. For **b_op**, it is the responsibility of the operation's caller to ensure that the code set tag parameters are set correctly before any data is marshalled.

IDL File

```
typedef byte my_byte;

void a_op(
  [in] unsigned long stag,
  [in] unsigned long drtag,
  [out] unsigned long *p_rtag,
  [in] long s,
  [in, out] long *p_l,
  [in, out, size_is(s), length_is(*p_l)] my_byte a[]
);

void b_op(
  [in] unsigned long stag,
  [in] unsigned long drtag,
  [out] unsigned long *p_rtag,
  [in] long s,
  [in, out] long *p_l,
  [in, out, size_is(s), length_is(*p_l)] my_byte a[]
);
```

ACF

```
typedef [cs_char(ltype)] my_byte;

[cs_tag_rtn(set_tags)] a_op([cs_stag] stag,
             [cs_drtag] drtag,
             [cs_rtag] p_rtag);

           b_op([cs_stag] stag,
              [cs_drtag] drtag,
              [cs_rtag] p_rtag);
```

Generated Header File

```
typedef byte my_byte;

void a_op(
  /* [in] */ idl_long_int s,
  /* [in, out] */ idl_long_int *p_l,
  /* [in, out, size_is(s), length_is(*p_l)] */ ltype a[]
);

void b_op(
  /* [in] */ idl_ulong_int stag,
  /* [in] */ idl_ulong_ing drtag,
  /* [out] */ idl_ulong_int *p_rtag,
  /* [in] */ idl_long_int s,
  /* [in, out] */ idl_long_int *p_l,
  /* [in, out, size_is(s), length_is(*p_l)] */ ltype a[]
);
```

The binding_callout Attribute

The **binding_callout** attribute permits you to specify the name of a routine that the client stub is to call automatically to modify a server binding handle before it initiates a remote procedure call. This attribute is intended for use by client applications that employ the automatic binding method through the **auto_handle** ACF interface attribute. In automatic binding, it is the client stub, rather than the client application code, that obtains the binding handle to the server. The **binding_callout** attribute allows a client application using automatic binding to modify the binding handle obtained by the client stub. Without this attribute, it is impossible for the client application to modify the binding handle before the client stub attempts to initiate a remote procedure call to the selected server.

Clients typically use this attribute to augment automatic binding handles with security context, for example, so that authenticated RPC is used between client and server.

The **binding_callout** attribute has the following syntax. (See the example at the end of this section.)

```
[binding_callout(routine_name)] interface interface_name
```

The *routine_name* specifies the name of a binding callout routine that the client stub will call to modify the server binding handle before initiating the remote procedure call to the server. The IDL compiler will generate a function prototype for *routine_name* in the generated header file.

You can specify the name of a routine that you supply, or you can specify the DCE RPC routine **rpc_ss_bind_authn_client()** to modify the binding handle if it meets the needs of your application. See the **rpc_ss_bind_authn_client(3rpc)** reference page for more information.

The binding callout routine you provide must have a specific routine signature. See the **rpc_ss_bind_authn_client(3rpc)** reference page for information about the required routine signature.

The **binding_callout** attribute can occur at most once in the ACF and applies to all of the operations in the corresponding IDL file.

A binding callout routine should return the **error_status_ok** status code when it successfully modifies the binding handle or determines that no action is necessary. This status code causes the client stub to initiate the remote procedure call.

A binding callout routine can also return error status. If it does, the client stub does not initiate the remote procedure call. Instead, if the **auto_handle** attribute has been applied in the ACF, the client stub attempts to locate another server of the interface and then calls the binding callout routine again. If **auto_handle** is not in use, the client stub invokes its normal error-handling logic. A binding callout routine for a client using **auto_handle** can return the status code **rpc_s_no_more_bindings** to prevent the client stub from searching for another server and instead invoking its error-handling logic immediately.

By default, the client stub handles an error condition by raising an exception. If a binding callout routine returns one of the **rpc_s_** status codes, the client stub raises a matching **rpc_x_** exception. However, if a binding callout routine returns any other type of status code, the client stub will most likely raise it as an “unknown status” exception.

If the **comm_status** parameter ACF attribute has been applied to an operation, the client stub handles an error condition by returning the error status value in the **comm_status** parameter. Consequently, a binding callout routine can return any error status value to the client application code if the **comm_status** attribute has been applied to the operation.

A binding callout routine can raise a user-defined exception, rather than return a status code, to report application-specific error conditions back to the client application code using exceptions.

Example Using the binding_callout Attribute

ACF

```
[auto_handle,binding_callout(my_bh_callout)] interface \
 bindcall
{
}
```

Generated Header File (bindcall.h)

```
void my_bh_callout(
   rpc_binding_handle_t *p_binding,
   rpc_if_handle_t interface_handle,
   error_status_t *p_st
);
```

The C++ Attributes cxx_new, cxx_static, cxx_lookup, and cxx_delegate

The IDL compiler uses an ACF to do the following for C++ applications:

- Declare a server's manager class and object constructor by using **cxx_new**.
- Declare interface member functions as static by using the **cxx_static** attribute, if they are not already declared in the interface definition file.
- Rename static member functions by using **cxx_static.**.
- Specify a lookup function by using the **cxx_lookup** attribute. The server calls this application-specific function automatically if a client requests a known object not currently maintained by the server.
- Specify a delegate interface class by using the **cxx_delegate** attribute. A third-party class is encapsulated by a delegate class so it can be used in RPCs without modifying the original class.
- Control in which stub files application-specific header files are included. (See also the **include** statement and the **cstub** and **sstub** attributes.)

Using cxx_new to Declare an Object Creator Function

Member functions may be specified as static object creator functions by applying the **cxx_new** attribute to the function name. An object creator function allows clients to dynamically create remote objects of an interface class. Servers require this feature to specify their implementation-specific manager class, and clients can use this feature to specify their local implementation of the interface class. The **cxx_new** attribute is applied to an operation and has the following format:

```
[cxx_new (manager_class) ] creator_function( );
```

The associated IDL file must contain a function that returns a pointer to the interface class and that matches the *creator_function* name.

The *manager_class* argument to the **cxx_new**attribute specifies the class name the application uses to locally implement the interface class. The server stub requires the *manager_class,*which must be declared for it in a header file and included by using the **sstub** attribute with the **include** statement. The format in C++ of the *manager_class* declaration is as follows:

```
class manager_class : public interface_class {
    // The constructor, destructor, function declarations, and data.
    ...
}
```

The manager class may include not only a constructor to create objects of the class, but also all the nonstatic member functions declared in the interface class. A server implements the *manager_class* member functions in its manager code. Manager code handles requests from clients for all of the interface's member functions (static and nonstatic), including the *creator_function* which dynamically creates interface objects on the server for the client.

When a client calls the *creator_function*, the client stub executes a proxy function that uses RPCs to execute a manager class constructor on a server. A client can use the *manager_class* argument to specify a class name when implementing its own local version of the interface class. In this case, the client links into its application the **idl**-generated server stub along with a local implementation of the interface class. When the client uses the **new** operator to creates objects of the *manager_class* type, the object is local and no RPCs are involved.

Using cxx_static to Specify Static Interface Member Functions

Member functions can be specified as static by applying the **cxx_static** attribute to the function name. The **cxx_static** attribute is an operation attribute with a format as follows:

```
[cxx_static [(local_function)] ] member_function( );
```

The *member_function* name must match a function named in the associated IDL file.

Both remote and local versions of objects are implemented in an application by linking in both client and server stubs. A client links in the server stub to implement client-local versions of interface objects. A server links in the client stub to allow access to remote (in this case, client-local) objects that are passed in as parameters to member functions. The *local_function* argument to the **cxx_static** attribute specifies the function name that the application uses to locally implement the static member function and avoid name conflicts with the remote version of the function (always defined in the client stub).

The server implements a static function named *member_function* in its manager code if no *local_function* is specified. However, if a *local_function* is specified, the server implements a static function named *local_function* in its manager code. In this case, the *manager_function* is automatically implemented in the client stub which is linked into the application (along with the server stub) to handle marshalling of parameters that are client- local objects (remote to the server). When there is a *local_function*, it must be declared for the server stub in a header file and included by using the **sstub** attribute with the **include** statement in the ACF.

When a client calls the static *member_function*, the client stub executes a proxy function that uses RPCs to execute the associated remote function on a server. A client uses the *local_function* argument to specify a function name to use when implementing its own local version of the *member_function*. In this case, in addition to linking in the client stub, the client application also links in the **idl**-generated server stub with a local implementation of the function. When you develop a client, it might be easier if you think of the server stub as a local-implementation stub, because, when the client calls the static *local_function*, the call is strictly local and no RPCs are involved. If you compile an interface such that a server stub is not generated, the *local_function* argument to **cxx_static** is ignored.

An interface can instead specify static member functions by using the **static** keyword in the interface definition, in front of a member function.

Using cxx_lookup to Declare a Server's Object Lookup Function

If a client requests the use of a known object that is not yet in the server's runtime, the server can automatically look it up and create it by using an application-specific function. The object lookup function name is specified by using the **cxx_lookup** attribute in an ACF header. The ACF format is as follows:

```
cxx_lookup (object_lookup_function)
```

The *object_lookup_function* must be declared in a header file and included in the server stub by using the **sstub** attribute with the **include** statement in the ACF. The C++ function declaration must have the following format:

```
interface_name *object_lookup_function(uuid_t *);
```

The lookup function has one input pointer argument of type **uuid_t** representing the UUID of the object desired. The function returns a pointer to the interface class. The returned pointer represents the newly created object. If the object cannot be found or created, the function must return a zero.

Summary of Attributes

The following table lists the attributes available for use in the ACF and where in the file the attribute can be used.

Table 15. Summary of the ACF Attributes

Attribute	Where Used
auto_handle	Interface header
binding_callout	Interface header
code	Interface header, operation
comm_status	Operation, parameter
cs_char	Type
cs_drtag	Parameter
cs_rtag	Parameter
cs_stag	Parameter
cs_tag_rtn	Operation, interface header
cxx_delegate	Interface header
cxx_lookup	Interface header
cxx_new	Operation
cxx_static	Operation
cstub	include statement
decode	Operation, interface header
enable_allocate	Operation

Table 15. Summary of the ACF Attributes (continued)

Attribute	Where Used
encode	Operation, interface header
explicit_handle	Interface header, operation
extern_exceptions	Interface header
fault_status	Operation, parameter
heap	Type, parameter
implicit_handle	Interface header
nocode	Interface header, operation
represent_as	Type
sstub	include statement

Attribute Configuration Language

This section summarizes the ACF syntax, in extended BNF notation.

```
<acf_interface> ::=
  <acf_interface_header> "{" <acf_interface_body> "}"

<acf_interface_header> ::=
  [ <acf_interface_attr_list> ] "interface" <idl_interface_name>

<acf_interface_attr_list> ::= "[" <acf_interface_attrs> "]"

<acf_interface_attrs> ::=
  <acf_interface_attr> [ "," <acf_interface_attr> ] ...

<acf_interface_attr> ::= <acf_code_attr>
  | <acf_nocode_attr>
  | <acf_auto_handle_attr>
  | <acf_explicit_handle_attr>
  | <acf_implicit_handle_attr>
  | <acf_cs_tag_rtn_attr>
  | <acf_extern_exceps_attr>
  | <acf_encode_attr>
  | <acf_decode_attr>
  | <acf_binding_callout_attr>
  | <acf_delegate_attr>
  | <acf_lookup_attr>

<acf_auto_handle_attr> ::= "auto_handle"

<acf_explicit_handle_attr> ::= "explicit_handle"

<acf_implicit_handle_attr> ::=
  "implicit_handle" "(" <acf_named_type> <Identifier> ")"

<acf_extern_exceps_attr> ::=
  "extern_exceptions" "(" <acf_ext_excep_list> ")"

<acf_ext_exceps_list> ::=
  "<acf_ext_excep> [","  <acf_ext_excep] ...

<acf_ext_excep> ::= <Identifier>
```

```
<acf_binding_callout_attr> ::=
  "binding_callout" "(" <acf_bind_call_rtn_name> ")"

<acf_delegate_attr> ::= "cxx_delegate" "(" acf_delegate_name ")"

<acf_delegate_name> ::= <Identifier>

<acf_lookup_attr> ::= "cxx_lookup" "(" acf_lookup_name ")"

<acf_lookup_name> ::= <Identifier>

<acf_bind_call_rtn_name> ::= <Identifier>

<acf_interface_name> ::= <Identifier>

<acf_interface_body> ::= [ <acf_body_element> ] ...

<acf_body_element> ::= <acf_include> ";"
  | <acf_type_declaration> ";"
  | <acf_operation> ";"

<acf_include> ::= [ <acf_include_attr> ] \
       "include" <acf_include_list>

<acf_include_attr> ::= "sstub" | "cstub" | "sstub" "," "cstub"

<acf_include_list> ::= <acf_include_name> \
       [ "," <acf_include_name> ] ...

<acf_include_name> ::= """ <filename> """

<acf_type_declaration> ::= typedef [ <acf_type_attr_list> ] \
                        <acf_named_type>

<acf_named_type> ::= <Identifier>

<acf_type_attr_list> ::= "[" <acf_type_attrs> "]"

<acf_type_attrs> ::= <acf_type_attr> [ "," <acf_type_attr> ] ...

<acf_type_attr> ::= <acf_represent_attr>
  | <acf_cs_char_attr>
  | <acf_heap_attr>

<acf_represent_attr> ::= "represent_as" "(" <acf_repr_type> ")"

<acf_cs_char_attr> ::=
 "cs_char" "C" "(" <acf_cs_char_type> ")"

<acf_cs_char_type> ::= <acf_named_type>

<acf_repr_type> ::= <acf_named_type>

<acf_operation> ::= [ <acf_op_attr_list> ] <Identifier> "("
  [ <acf_parameters> ] ")"

<acf_op_attr_list> ::= "[" <acf_op_attrs> "]"
```

```
<acf_op_attrs> ::= <acf_op_attr> [ "," <acf_op_attr> ] ...

<
acf_op_attr> ::= <acf_explicit_handle_attr>
    | <acf_comm_status_attr>
    | <acf_cs_tag_rtn_attr>
    | <acf_encode_attr>
    | <acf_decode_attr>
    | <acf_fault_status_attr>
    | <acf_code_attr>
    | <acf_nocode_attr>
    | <acf_enable_allocate_attr>
    | <acf_static_attr>
    | <acf_new_attr>

<acf_cs_tag_rtn_attr> ::=
 "cs_tag_rtn" "(" <acf_cs_tag_rtn_name> ")"

<acf_cs_tag_rtn_name> ::=
 <Identifier>

<acf_parameters> ::= <acf_parameter> [ "," <acf_parameter> ] ...

<acf_parameter> ::= [ <acf_param_attr_list> ] <Identifier>

<acf_param_attr_list> ::= "[" <acf_param_attrs> "]"

<acf_param_attrs> ::= <acf_param_attr> [ "," <acf_param_attr> ] ...

<acf_param_attr> ::= <acf_comm_status_attr>
    | <acf_fault_status_attr>
    | <acf_cs_stag_attr>
    | <acf_cs_drtag_attr>
    | <acf_cs_rtag_attr>
    | <acf_heap_attr>

<acf_code_attr> ::= "code"

<acf_nocode_attr> ::= "nocode"

<acf_encode_attr> ::= "encode"

<acf_decode_attr> ::= "decode"

<acf_cs_stag_attr> ::= "cs_stag"

<acf_cs_drtag_attr> ::= "cs_drtag"

<acf_cs_rtag_attr> ::= "cs_rtag"

<acf_comm_status_attr> ::= "comm_status"

<acf_fault_status_attr> ::= "fault_status"

<acf_enable_allocate_attr> ::= "enable_allocate"

<acf_static_attr> ::= "cxx_static" | "cxx_static" \
     "(" <acf_static_name> ")"

<acf_static_name> ::= <Identifier>
```

```
<acf_new_attr> ::= "cxx_new" "(" <acf_new_name> ")"
<acf_new_name> ::= <Identifier>

<acf_heap_attr> ::= "heap"
```

Part 4. DCE Distributed Time Service

Chapter 20. Introduction to the Distributed Time Service API

This chapter describes the DCE Distributed Time Service (DTS) programming routines. You can use these routines to obtain timestamps that are based on Coordinated Universal Time (UTC). You can also use the DTS routines to translate among different timestamp formats and perform calculations on timestamps. Applications can use the timestamps that DTS supplies to determine event sequencing, duration, and scheduling. Applications can call the DTS routines from any host that has the **libdce**. The **dtsd** need not be running.

DTS routines are written in the C programming language. You should be familiar with basic DTS concepts before you attempt to use the application programming interface (API). The DTS chapters of the *OSF DCE Administration Guide—Core Components* provide conceptual information about DTS.

The DTS API routines offer the following basic functions:
- Retrieving timestamp information
- Converting between binary timestamps that use different time structures
- Converting between binary timestamps and ASCII representations
- Converting between UTC time and local time
- Manipulating binary timestamps
- Comparing two binary time values
- Calculating binary time values
- Obtaining time zone information

The sections that follow describe how DTS represents time, discuss the DTS time structures, discuss the DTS API header files, and briefly describe the DTS API routines.

DTS Time Representation

UTC is the international time standard that has largely replaced Greenwich Mean Time (GMT). The standard is administered by the International Time Bureau (BIH) and is widely used. DTS uses opaque binary timestamps that represent UTC for all of its internal processes. You cannot read or disassemble a DTS binary timestamp; the DTS API allows applications to convert or manipulate timestamps, but they cannot be displayed. DTS also translates the binary timestamps into ASCII text strings, which can be displayed.

Absolute Time Representation

An absolute time is a point on a time scale. For DTS, absolute times reference the UTC time scale; absolute time measurements are derived from system clocks or external time-providers. When DTS reads a system clock time, it records the time in an opaque binary timestamp that also includes the inaccuracy and other information. When you display an absolute time, DTS converts the time to ASCII text as shown in the following display:

```
1990-11-21-13:30:25.785-04:00I000.082
```

DTS displays all times in a format that complies with the International Organization for Standardization (ISO) 8601 (1988) standard. Note that the inaccuracy portion of the time is not defined in the ISO standard; times that do not include an inaccuracy are accepted.

Figure 59 explains the ISO format that generated the previous display.

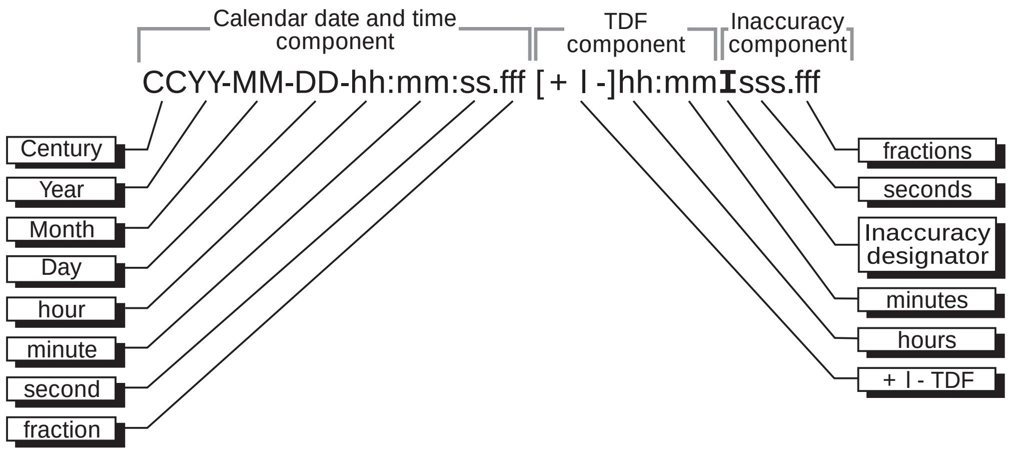


Figure 59. ISO Format for Time Displays

In this figure, the relative time preceded by the **+** (plus) or **−** (minus) character indicates the hours and minutes that the calendar date and time are offset from UTC. The presence of this time differential factor (TDF) in the string also indicates that the calendar date and time are the local time of the system, not UTC. Local time is UTC plus the TDF. The Inaccuracy (**I**) designator indicates the beginning of the inaccuracy component associated with the time.

Although DTS displays all times in the previous format, variations to the ISO format shown in Figure 60 are also accepted as input for the ASCII conversion routines.

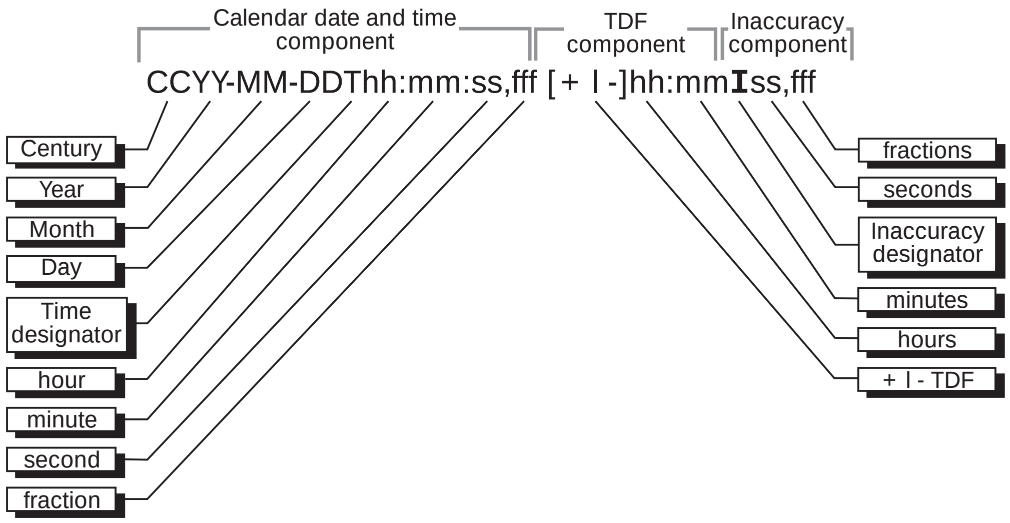


Figure 60. Variations to the ISO Time Format

In this figure, the Time (**T**) designator separates the calendar date from the time, a **,** (comma) separates seconds from fractional seconds, and the **+** or **−** indicates the beginning of the inaccuracy component.

The following examples show some valid time formats.

The following represents July 4, 1776 17:01 GMT and an unspecified inaccuracy (default):

```
1776-7-4-17:01:00
```

The following represents a local time of 12:01 (17:01 GMT) on July 4, 1776 with a TDF of −5 hours and an inaccuracy of 100 seconds:

```
1776-7-4-12:01:00-05:00I100
```

Both of the following represent 12:00 GMT in the current day, month, and year with an unspecified inaccuracy:

```
12:00 and T12
```

The following represents July 14, 1792 00:00 GMT with an unspecified inaccuracy:

```
1792-7-14
```

Relative Time Representation

A relative time is a discrete time interval that is usually added to or subtracted from another time. A TDF associated with an absolute time is one example of a relative time. A relative time is normally used as input for commands or system routines.

Figure 61 shows the full syntax for a relative time.

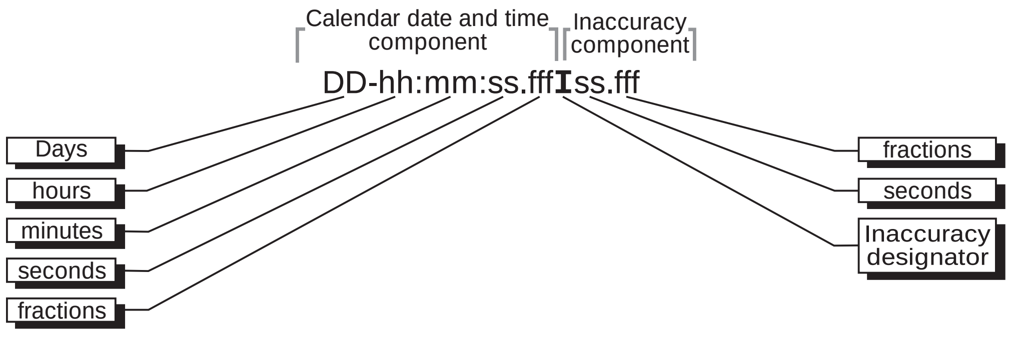


Figure 61. Full Syntax for a Relative Time

The following example shows a relative time of 21 days, 8 hours, and 30 minutes, 25 seconds with an inaccuracy of 0.300 seconds:

```
21-08:30:25.000I00.300
```

The following example shows a negative relative time of 20.2 seconds with an unspecified inaccuracy (default):

```
-20.2
```

The following example shows a relative time of 10 minutes, 15.1 seconds with an inaccuracy of 4 seconds:

```
10:15.1I4
```

Notice that a relative time does not use the calendar date fields, since these fields concern absolute time. A positive relative time is unsigned; a negative relative time is preceded by a **−** (minus) sign. A relative time is often subtracted from or added to another relative or absolute time. Relative times that DTS uses internally are opaque binary timestamps. The DTS API offers several routines that can be used to calculate new times by use of relative binary timestamps.

Representing Periods of Time

A given duration of a period of time can be represented by a data element of variable length that uses the syntax shown in Figure 62.

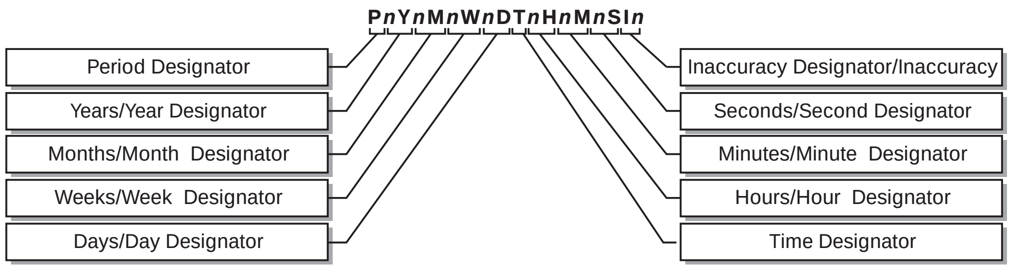


Figure 62. Syntax for Representing a Duration

The Data Element Parts

The data element contains the following parts:

- The designator **P** precedes the part that includes the calendar components, including the following:
  - The number of years followed by the designator **Y**
  - The number of months followed by the designator **M**
  - The number of weeks followed by the designator **W**
  - The number of days followed by the designator **D**
- The **T** designator precedes the part that includes the time components, including the following:
  - The number of hours followed by the designator **H**
  - The number of minutes followed by the designator **M**
  - The number of seconds followed by the designator **S**
- The designator **I** precedes the number of seconds of inaccuracy.

The following example represents a period of 1 year, 6 months, 15 days, 11 hours, 30 minutes, and 30 seconds and an unspecified inaccuracy:

```
P1Y6M15DT11H30M30S
```

The following example represents a period of 3 weeks and an inaccuracy of 4 seconds:

```
P3WI4
```

Time Structures

DTS can convert among several types of binary time structures that are based on different base dates and time unit measurements. DTS uses UTC-based time structures and can convert other types of time structures to its own presentation of UTC-based time. The DTS API routines are used to perform these conversions for applications on your system.

Table 16 lists the absolute time structures that the DTS API uses to modify binary times for applications.

Table 16. Absolute Time Structures

Structure	Time Units	Base Date	Approximate Range
utc	100-nanosecond	15 October 1582	A.D. 1 to A.D. 30,000
tm	second	1 January 1900	A.D. 1 to A.D. 30,000
timespec	nanosecond	1 January 1970	A.D. 1970 to A.D. 2106

Table 17 lists the relative time structures that the DTS API uses to modify binary times for applications.

Table 17. Relative Time Structures

Structure	Time Units	Approximate Range
utc	100-nanosecond	+/− 30,000 years
tm	second	+/- 30,000 years
reltimespec	nanosecond	+/- 68 years

The remainder of this section explains the DTS time structures in detail.

The utc Structure

UTC is useful for measuring time across local time zones and for avoiding the seasonal changes (summer time or daylight savings time) that can affect the local time. DTS uses 128-bit binary numbers to represent time values internally; throughout this guide, these binary numbers representing time values are referred to as binary timestamps. The DTS **utc** structure determines the ordering of the bits in a binary timestamp; all binary timestamps that are based on the **utc** structure contain the following information:

- The count of 100-nanosecond units since 00:00:00.00, 15 October 1582 (the date of the Gregorian reform to the Christian calendar)
- The count of 100-nanosecond units of inaccuracy applied to the preceding item
- The TDF, expressed as the signed quantity
- The DTS version number

The binary timestamps that are derived from the DTS **utc** structure have an opaque format. This format is a cryptic character sequence that DTS uses and stores internally. The opaque binary timestamp is designed for use in programs, protocols, and databases.

Note: Applications use the opaque binary timestamps when storing time values or when passing them to DTS.

The API provides the necessary routines for converting between opaque binary timestamps and character strings that can be displayed and read by users.

The tm Structure

The **tm** structure is based on the time in years, months, days, hours, minutes, and seconds since 00:00:00 GMT (Greenwich Mean Time), 1 January 1900. The **tm** structure is defined in the **time.h** header file.

The **tm** structure declaration follows:

```
struct tm {
   int tm_sec;  /* Seconds (0 - 59)          */
   int tm_min;  /* Minutes (0 - 59)          */
   int tm_hour;  /* Hours (0 - 23)           */
   int tm_mday;  /* Day of Month (1 - 31)       */
   int tm_mon;  /* Month of Year (0 - 11)       */
   int tm_year;  /* Year - 1900           */
   int tm_wday;  /* Day of Week (Sunday = 0)      */
   int tm_yday;  /* Day of Year (0 - 364)       */
   int tm_isdst; /* Nonzero if Daylight Savings Time  */
          /* is in effect           */
     };
```

Not all of the **tm** structure fields are used for each routine that converts between **tm** structures and **utc** structures. (See the parameter descriptions contained in the reference pages in the *OSF DCE Application Development Reference* for additional information about which fields are used for specific routines.)

The timespec Structure

The **timespec** structure is normally used in combination with or in place of the **tm** structure to provide finer resolution for binary times. The **timespec** structure is similar to the **tm** structure, but the **timespec** structure specifies the number of seconds and nanoseconds since the base time of 00:00:00 GMT, 1 January 1970. You can find the structure in the **dce/utc.h** header file.

The **timespec** structure declaration follows:

```
struct timespec {
  time_t tv_sec;    /* Seconds since 00:00:00 GMT, */
             /*  1 January 1970       */
  long  tv_nsec;   /* Additional nanoseconds since */
             /*  tv_sec           */
       }  timespec_t;
```

The reltimespec Structure

The **reltimespec** structure represents relative time. This structure is similar to the **timespec** structure, except that the first field is *signed* in the **reltimespec** structure. (The field is *unsigned* in the **timespec** structure.) You can find the **reltimespec** structure in the **dce/utc.h** header file.

The **reltimespec** structure declaration follows:

```
struct reltimespec {
     time_t tv_sec;  /* Seconds of relative time  */
     long  tv_nsec; /* Additional nanoseconds of  */
             /*  relative time       */
       }  reltimespec_t;
```

DTS API Header Files

The **time.h** and **dce/utc.h** header files contain the data structures, type definitions, and define statements that are referenced by the DTS API routines. The **time.h** header file is a standard UNIX file. The **dce/utc.h** header file includes **time.h** and contains the **timespec**, **reltimespec**, and **utc** structures.

These header files are located in **/usr/include/dce**.

DTS API Routine Functions

Figure 63 categorizes the DTS portable interface routines by function.

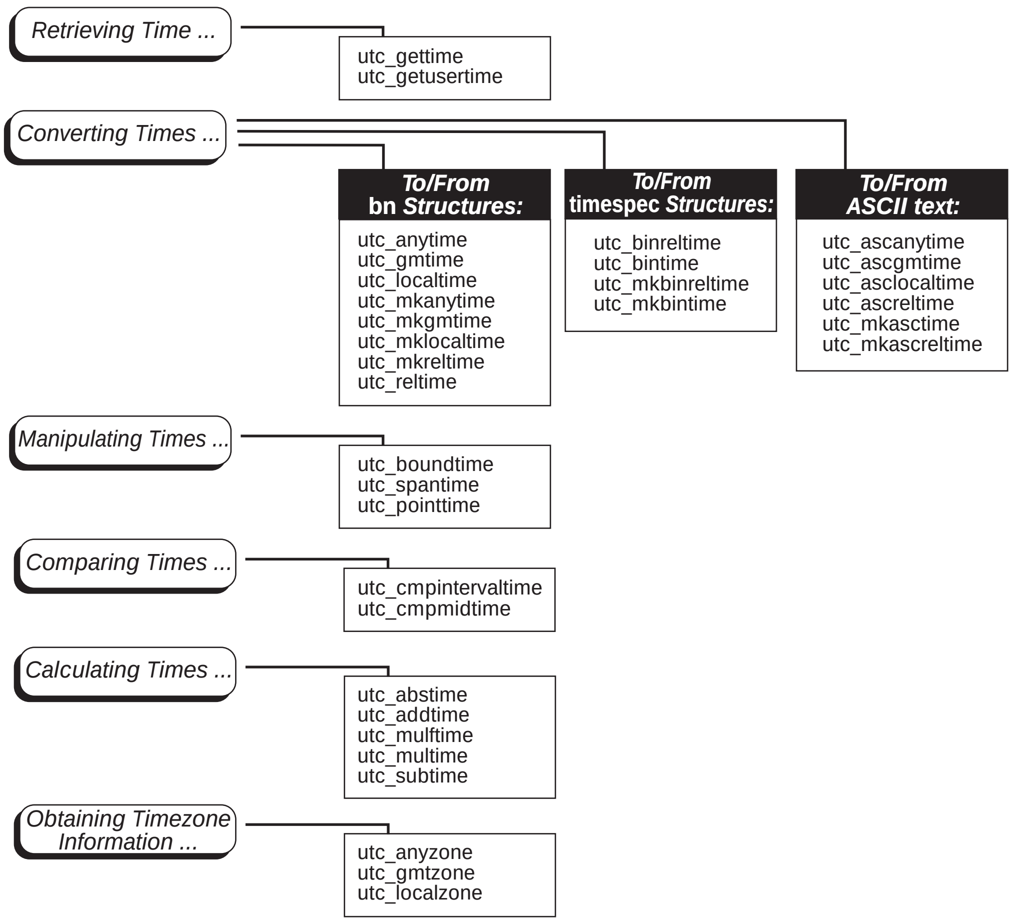


Figure 63. DTS API Routines Shown by Functional Grouping

An alphabetical listing of the DTS portable interface routines and a brief description of each one follows:

- **utc_abstime**: Computes the absolute value of a binary relative timestamp
- **utc_addtime**: Computes the sum of two binary timestamps; the timestamps can be two relative times or a relative time and an absolute time
- **utc_anytime**: Converts a binary timestamp into a **tm** structure by using the TDF information contained in the timestamp to determine the TDF returned with the **tm** structure
- **utc_anyzone**: Gets the time zone label and offset from GMT by using the TDF contained in the input **utc**
- **utc_ascanytime**: Converts a binary timestamp into an ASCII string that represents an arbitrary time zone
- **utc_ascgmtime**: Converts a binary timestamp into an ASCII string that expresses a GMT time
- **utc_asclocaltime**: Converts a binary timestamp to an ASCII string that represents a local time
- **utc_ascreltime**: Converts a binary timestamp that expresses a relative time to its ASCII representation
- **utc_binreltime**: Converts a relative binary timestamp into two **timespec** structures that express relative time and inaccuracy
- **utc_bintime**: Converts a binary timestamp into a **timespec** structure
- **utc_boundtime**: Given two UTC times, one before and one after an event, returns a single UTC time whose inaccuracy includes the event
- **utc_cmpintervaltime**: Compares two binary timestamps or two relative binary timestamps
- **utc_cmpmidtime**: Compares two binary timestamps or two relative binary timestamps, ignoring inaccuracies
- **utc_gettime**: Returns the current system time and inaccuracy as an opaque binary timestamp
- **utc_getusertime**: Returns the time and process-specific TDF, rather than the system-specific TDF
- **utc_gmtime**: Converts a binary timestamp into a **tm** structure that expresses GMT or the equivalent UTC
- **utc_gmtzone**: Gets the time zone label, given **utc**
- **utc_localtime**: Converts a binary timestamp into a **tm** structure that expresses local time
- **utc_localzone**: Gets the time zone label and offset from GMT, given **utc**
- **utc_mkanytime**: Converts a **tm** structure and TDF (expressing the time in an arbitrary time zone) into a binary timestamp
- **utc_mkascreltime**: Converts a null-terminated character string, which represents a relative timestamp, to a binary timestamp
- **utc_mkasctime**: Converts a null-terminated character string, which represents an absolute timestamp, to a binary timestamp
- **utc_mkbinreltime**: Converts a **timespec** structure expressing a relative time to a binary timestamp
- **utc_mkbintime**: Converts a **timespec** structure into a binary timestamp
- **utc_mkgmtime**: Converts a **tm** structure that expresses GMT or UTC to a binary timestamp
- **utc_mklocaltime**: Converts a **tm** structure that expresses local time to a binary timestamp

- **utc_mkreltime**: Converts a **tm** structure that expresses relative time to a binary timestamp
- **utc_mulftime**: Multiplies a relative binary timestamp by a floating-point value
- **utc_multime**: Multiplies a relative binary timestamp by an integer factor
- **utc_pointtime**: Converts a binary timestamp to three binary timestamps that represent the earliest, most likely, and latest time
- **utc_reltime**: Converts a binary timestamp that expresses a relative time into a **tm** structure
- **utc_spantime**: Given two (possibly unordered) binary timestamps, returns a single UTC time interval whose inaccuracy spans the two input timestamps
- **utc_subtime**: Computes the difference between two binary timestamps that express either an absolute time and a relative time, two relative times, or two absolute times

Chapter 21. Time-Provider Interface

This chapter describes the Time-Provider Interface (TPI) for DCE Distributed Time Service software. The chapter provides a brief overview of the TPI, explains how to use external time-providers with DTS, and describes the data structures and message protocols that make up the TPI.

Coordinated Universal Time (UTC) is widely used and is disseminated throughout the world by various standards organizations. Several manufacturers supply devices that can acquire UTC time values via radio, satellite, or telephone. These devices can then provide standardized time values to computer systems. Normally, one device is connected to a computer system; the device runs a process that interprets signals and translates them to time values, which can either be displayed or be provided to the server process running on the connected system.

To synchronize its system clock with UTC using an external time-provider device, a DTS server needs a software interface to the device to periodically obtain UTC. In effect, this interface serves as an intermediary between the DTS server and external time-provider processes. The DTS server requires the interface to obtain UTC time values and to determine the associated inaccuracy of each value. The interface between the DTS server process and the time-provider process is called the Time-Provider Interface.

The remainder of this chapter describes the TPI and its attendant processes in detail. The following section describes the control flow between the DTS server process, the TPI, and the time-provider process.

General TPI Control Flow

When you use a time-provider with a system running DTS, the external time-provider is implemented as an independent process that communicates with a DTS server process through remote procedure calls (RPCs). A remote procedure call is a synchronous request and response between a main calling program and a procedure executing in another process. RPC applications are based on the client/server model. In this context, the following processes act as the client and server components in the RPC-based application:

- The DTS daemon is the client.
- The Time-Provider process (TP process) is the server.

Both the RPC-client (DTS daemon) and the server (TP process) must be running on the same system.

Applications running on RPC communicate through an interface that is well known to both the client and the server. The RPC interface consists of a set of procedures, data types, and constants that describe how a client can invoke a routine running on the server. The server offers the interface to the clients through the Interface Definition Language (IDL) file.

The IDL file defines the syntax for an operation, including the following:

- The name of the operation
- The data type of the value that the operation returns (if any)
- The order and data types of the operation’s parameters (if any)

The TP process offers two procedures that DTS calls to obtain time values. These procedures are **ContactProvider** and **ServerRequestProviderTime**.

At each system synchronization, DTS makes the initial remote procedure call (**ContactProvider**) to a TP process that is assumed to be running on the same node.

If the TP process is active, the RPC call returns the following arguments:

- A successful communication status message
- A control message that DTS uses for further processing

If the TP process is not active, the RPC call either returns a communication status failure or a time-out occurs. DTS then synchronizes with other servers instead of with the external time-provider.

If the initial call (**ContactProvider**) is successful, DTS makes a second call (**ServerRequestProviderTime**) to retrieve the timestamps from the external time-provider. The control message sent by the TP process in the first RPC call specifies the length of time DTS waits for the RPC call to complete. The TP process returns the following parameters in the procedure call:

- A communication status message.
- A time structure that contains timestamps collected from the external time-provider. (DTS then uses these timestamps to complete its synchronization.)

Figure 64 on page 471 illustrates the RPC calling sequence between DTS and the TP process. Note that solid black lines represent the path followed by input parameters; dashed lines represent the path followed by output parameters and return values.

The following steps describe the process shown in Figure 64 on page 471:

1. At synchronization time, DTS calls the **ContactProvider** remote procedure. Input parameters are passed to the TP client stub, dispatched to the RPC runtime library, and then passed to the TP server stub.
2. The TP process receives the call and executes the **ContactProvider** procedure.
3. The procedure terminates and returns the results through the TP server stub, the RPC runtime library, and the TP client stub.
4. The procedure terminates in the DTS call, where the returned parameters are examined.
5. DTS then calls the **ServerRequestProviderTime** remote procedure. Input parameters are passed to the TP client stub, dispatched to the RPC runtime library, and then passed to the TP server stub.

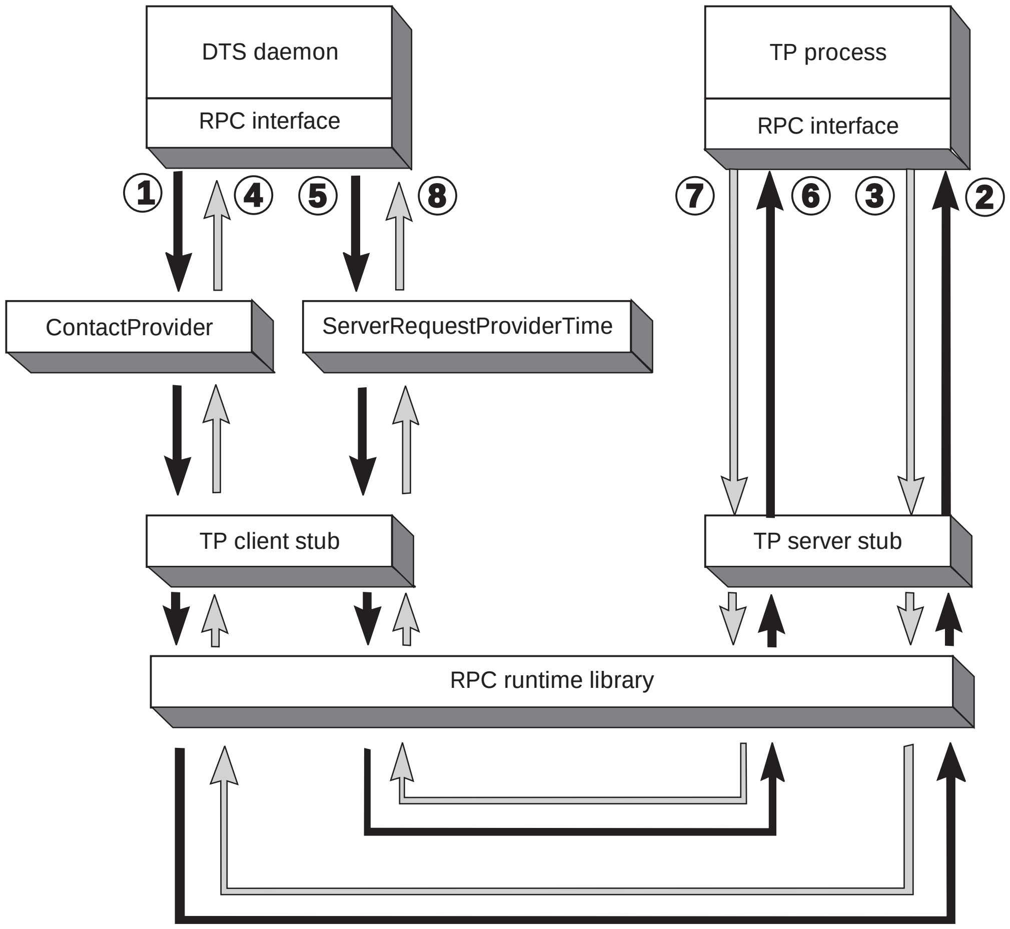


Figure 64. DTS/Time-Provider RPC Calling Sequence

6. The TP process receives the call and executes the **ServerRequestProviderTime** procedure.
7. The procedure terminates and returns the results through the TP server stub, the RPC runtime library, and the TP client stub.
8. The DTS remote procedure call terminates and the timestamps are returned as an output parameter. DTS then synchronizes using the timestamps returned by the external time-provider.

The following section describes the remote procedures that are exported by the TP process during the previous sequence.

ContactProvider Procedure

ContactProvider is the first routine called by DTS. The routine is called to verify that the TP process is running and to obtain a control message that DTS uses for subsequent communications with the TP process and for synchronization after it receives the timestamps. The parameters passed in the **ContactProvider** procedure call consist of the following elements:

- Binding Handle

  An input parameter that establishes the relationship between DTS and the TP process. A binding handle enables the client (DTS) to recognize and find a server (the TP process) that offers the same interface.

- Control Message

  An output parameter that contains information used by DTS for subsequent processing. The control message consists of the following elements:

  TPstatus
  : One of the following values:
    - **K_TPI_SUCCESS**
    - **K_TPI_FAILURE**

  nextPoll
  : A time value that tells DTS when to contact the TP process again. For example, once a day through dial-up, radio, or satellite.

  timeout
  : A value that tells DTS how long to wait for a response from the TP process.

  noClockSet
  : A value that specifies whether or not DTS is allowed to alter the system clock. If *noClockSet* is specified as 0x01 (TRUE), DTS does not adjust or set the clock during the current synchronization. This option is useful for systems whose system clock is known to be accurate (such as systems equipped with special hardware) or systems that are managed by some other time service (such as Network Time Protocol (NTP)), but which still wish to function as a DTS server.

- Communication Status

  An output parameter that contains a status code returned by the DCE RPC runtime library. The status **rpc_s_ok** is returned if the TP process is successfully contacted.

ServerRequestProviderTime Procedure

After the TP process is successfully contacted, DTS makes the **ServerRequestProviderTime** procedure call to obtain the timestamps from the external time-provider. The parameters passed in the **ServerRequestProviderTime** procedure call consist of the following elements:

- Binding Handle

  An input parameter that establishes the relationship between DTS and the TP process. A binding handle enables the client (DTS) to recognize and find a server (the TP process) that offers the same interface.

- Time Response Message

  An output parameter that contains a TP process status value (**K_TPI_SUCCESS** or **K_TPI_FAILURE**), a count of the timestamps that are returned, and the timestamps obtained from the external time-provider. The timestamp count is an integer in the range **K_MIN_TIMESTAMPS** to **K_MAX_TIMESTAMPS**. Each timestamp consists of three **utc** time values:

  - The system clock time immediately before the TP process polls the external time source. (The TP process normally obtains the time from the **utc_gettime()** DTS API routine.)
  - The time value returned to the TP process by the external time source.
  - The system clock time immediately after the external time source is read. (The TP process obtains the time from the **utc_gettime()** DTS API routine.)

- Communication Status

An output parameter that contains a status code returned by the DCE RPC runtime library. The status **rpc_s_ok** is returned if the TP process is successfully contacted.

Time-Provider Process IDL File

A remote procedure call can only work if an interface definition that clearly defines operation signatures exists. Operation signatures define the syntax for an operation, including its name and parameters (input and output) that are passed as part of the procedure call. The TP process interface exports the two operation signatures that have been previously explained. The interface is located in the file **examples/dts/dtsprovider.idl**. When building the TP process application, this file must be compiled using the IDL compiler, which creates three files:

- **dtsprovider.h** (header file)
- **dtsprovider_sstub.c** (server stub file)
- **dtsprovider_cstub.c** (client stub file)

The Time-Provider program (TP program) must be compiled along with the **dtsprovider_sstub.c** code and then linked together. The TP program must also include the stub-generated file **dtsprovider.h**. The following sample code shows the structure of this interface.

```
/*
 *       Time Service Provider Interface
 *
 * This interface is defined through the Network Interface
 * Definition Language (NIDL).
 */
[uuid (bfca1238-628a-11c9-a073-08002b0dea7a),
  version(1)
]
interface time_provider
{

 import "dce/nbase.idl";
 import "dce/utctypes.idl";

/* Minimum and Maximum number of times to read time source at
 * each synchronization
 */
const long K_MIN_TIMESTAMPS  = 1;
const long K_MAX_TIMESTAMPS  = 6;

/* Message status field return values
 */
const long K_TPI_FAILURE    = 0;
const long K_TPI_SUCCESS    = 1;

/* This structure contains one reading of the TP wrapped in
 * the timestamps of the local clock.
 */
typedef struct TimeResponseType
{
  utc_t beforeTime; /* local clk just before getting UTC */
  utc_t TPtime;   /* source UTC; inacc also supplied  */
  utc_t afterTime;  /* local clk just after getting UTC */
} TimeResponseType;

/* Time-provider control message. This structure is returned
 * in response to a time service request. The status field
 * returns TP success or failure. The nextPoll gives the
 * client the time at which to poll the TP next. The timeout
 * value tells the client how long to wait for a time response
 * from the TP. The noClockSet will tell the client whether
 * or not it is allowed to alter the system clock after a
 * synchronization with the TP.
 */
typedef struct TPctlMsg
{
  unsigned long    status;
  unsigned long    nextPoll;
  unsigned long    timeout;
  unsigned long    noClockSet;
} TPctlMsg;
/* TP timestamp message. The actual time-provider
 * synchronization data. The status is the result of the
 * operation (success or failure). The timeStampCount
 * parameter returns the number of timestamps being returned
 * in this message. The timeStampList is the set of
 * timestamps being returned from the TP.
 */
typedef struct TPtimeMsg
{
  unsigned long    status;
  unsigned long    timeStampCount;
  TimeResponseType  timeStampList[K_MAX_TIMESTAMPS];

} TPtimeMsg;

/* The Time-Provider Interface structures are described here.
 * There are two types of response messages from the TP:
 * control message and data message.
 *
 *     <<<< TPI CONTROL MESSAGE >>>>
 *
 * 31                      0
 * +-------------------------------------------+
 * |      Time-Provider Status      |
 * +-------------------------------------------+
 * |      Next Poll Delta         |
 * +-------------------------------------------+
 * |      Message Time Out        |
 * +-------------------------------------------+
 * |      NoSet Flag           |
 * +-------------------------------------------+
 *
 *   <<<< a single timestamp >>>>
 *
 * 128                     0
 * +-------------------------------------------+
 * |      Before Time           |
 * +-------------------------------------------+
 * |      TP Time              |
 * +-------------------------------------------+
 * |      After Time            |
 * +-------------------------------------------+
 *
 *   <<<< TPI DATA MESSAGE >>>>
 *
 * 31                      0
 * +-------------------------------------------+
 * |      Time-Provider Status      |
 * +-------------------------------------------+
 * |      Timestamp Count         |
 * +-------------------------------------------+
 * |                       |
 * |      <timestamp one>         |
 * |                       |
 * +-------------------------------------------+
 * |         .           |
 * |         .           |
 * |         .           |
 * |         .           |
 * |         .           |
 * +-------------------------------------------+
 * |                       |
 * |   <timestamp K_MAX_TIMESTAMPS>     |
 * |                       |
 * +-------------------------------------------+
 */

/* The RPC-based Time-Provider Program (TPP) interfaces are
 * defined here. These calls are invoked by a Time Service
 * daemon running as a server (in this case it makes an RPC
 * client call to the TPP server).
 */
/* CONTACT_PROVIDER
 * Send initial contact message to the TPP. The TPP server
 * responds with a control message.
 */
void ContactProvider
    (
    [in]  handle_t    bind_h,
    [out]  TPctlMsg    *ctrlRespMsg,
    [out]  error_status_t *comStatus
    );

/* SERVER_REQUEST_PROVIDER_TIME
 * The client sends a request to the TPP for times. The
 * TPP server responds with an array of timestamps obtained
 * by querying the Time-Provider hardware that it polls.
 */
void ServerRequestProviderTime
    (
    [in]  handle_t    bind_h,
    [out]  TPtimeMsg    *timesRspMsg,
    [out]  error_status_t *comStatus
    );
}
```

Initializing the Time-Provider Process

Initializing the RPC-based TP process prepares it to receive remote procedure calls from a DTS daemon requesting the timestamps. The following steps are involved:

1. Include the header file (**dtsprovider.h**) that is created by compiling **/usr/include/dce/dtsprovider.idl**, which contains the interface definition.
2. Register the interface with the DCE RPC runtime.
3. Select one or more protocol sequences that are compatible with both the interface and the runtime library. It is recommended that the TP process application selects all protocol sequences available on the system. Available protocol sequences are obtained by calling an RPC API routine, described in the example that follows. This ensures that transport independence is maintained in RPC applications.
4. Register the TP process with the endpoint mapper service of the DCE daemon (**dced**) running on the system. This makes the TP process available to the DTS daemon.

5. Obtain the name of the machine's principal and then register an authentication service to use with authenticated remote procedure calls coming from the DTS daemon. Note that DTS and the TP program are presumed to be running in an authenticated environment.
6. Listen for remote procedure calls.

The following shows these steps, including the sequence of calls needed:

```
/* Register the TP server interface with the RPC runtime.
 * The interface specification time_provider_v1_0_ifspec
 * is obtained from the generated header file dtsprovider.h
 * The entry point vector is normally defined at the top of
 * the TP source program similar to this:
 *
 *  globaldef time_provider_v1_0_epv_t time_provider_epv =
 *  {
 *    ContactProvider,
 *    ServerRequestProviderTime
 *  };
 */
rpc_server_register_if (time_provider_v1_0_s_ifspec,
            NULL,
            (rpc_mgr_epv_t) &time_provider_epv,
            &RPCstatus);

/*
 * This call tells the DCE RPC runtime to listen for remote
 * procedure calls using all supported protocol sequences.
 * To listen for a specific protocol sequence, use the
 * rpc_server_use_protreq call.
 */
rpc_server_use_all_protseqs (max_calls,
               &RPCstatus);

/* This routine is called to obtain a vector of binding
 * handles that were established with registration of
 * protocol sequences.
 */
rpc_server_inq_bindings (&bind_vector,
             &RPCstatus);

/* This routine adds the address information of the binding
 * handle for the TP server to the endpoint mapper database.
 */
rpc_ep_register (time_provider_v1_0_s_ifspec,
         bind_vector,
         NULL,
         "Time-Provider",
         &RPCstatus);


/* Obtain the name of the machine's principal and register an
 * authentication service to use for authenticated remote
 * procedure calls coming from the time service daemon.
 */
dce_cf_prin_name_from_host (NULL,
              &machinePrincipalName,
              &status);

rpc_server_register_auth_info (machinePrincipalName,
               rpc_c_authn_dce_private,
               NULL,
               NULL,
               &RPCstatus);

/* This routine is called to listen for remote procedure calls
```

```
 * sent by the DTS client. Possible RPC calls coming from DTS
 * client are ContactProvider and ServerRequestProviderTime.
 */
rpc_server_listen (max_calls,
         &RPCstatus);
```

Time-Provider Algorithm

The time-provider algorithm assumes that the two remote procedure calls will come in the following order: **ContactProvider** followed by **ServerRequestProviderTime**. The algorithm to create a generic time-provider follows:

1. Initialize the TP process, as described previously. Listen for RPC calls.
2. If the **ContactProvider** procedure is invoked, perform the following steps:
   a. Initialize the control message to the appropriate values (status value to **K_TPI_SUCCESS**; *nextPoll*, *timeout*, and *noClockSet* to valid integer values).
   b. Set the communication status output parameter to **rpc_s_ok**.
   c. Return from the procedure call. (The DCE RPC runtime returns the values to DTS.)
3. If the **ServerRequestProviderTime** procedure is run, perform the following steps:
   a. Initialize the timestamp count to the appropriate number.
   b. Use the **utc_gettime()** DTS API routine to read the system time.
   c. Poll the external time source and read a UTC value. Use the **utc_gmtime()** routine to convert the UTC time value to a binary timestamp.
   d. Use the **utc_gettime()** routine to read the system time.
   e. Repeat steps b, c, and d the number of times specified by the values of **K_MIN_TIMESTAMPS** and **K_MAX_TIMESTAMPS**.
   f. If steps b, c, or d return erroneous data, initialize the TP process status field (*TPstatus*) of the data message to **K_TPI_FAILURE**; otherwise, initialize the data message timestamps.
   g. Set the communication status output parameter to **rpc_s_ok**.
   h. Return from the procedure call. (The DCE RPC runtime sends the values back to DTS.)
4. The TP process continues listening for RPC calls.

DTS Synchronization Algorithm

DTS performs the following steps to synchronize with an external time-provider:

1. At startup time, creates the binding handle for the TPI. The binding handle is obtained from the list of available protocol sequences on the system.
2. At synchronization time, makes the remote procedure call **ContactProvider**, assuming that a TP process is running on the system. If the procedure call fails, examine the RPC communication status, checking the availability of the server. If the server is unavailable, synchronize with peer servers; otherwise, continue.
3. Waits for the procedure call to return the control message in the output parameter. If the procedure call does not return within the specified LAN timeout interval, synchronizes with peer servers. Otherwise, go to step 4.
4. If the procedure call returned successfully (communication status is **rpc_s_ok**), reads the data in the control message.

5. Makes the remote procedure call **ServerRequestProviderTime** to obtain the timestamps from the external time-provider. If the procedure does not return within the elapsed time specified by the control message (*timeout*), then synchronizes with peer servers. Schedules the next synchronization based upon the applicable DTS management parameters, ignoring *nextPoll*.
6. If the procedure returns successfully, verifies that the TP process status is **K_TPI_SUCCESS**. Otherwise, synchronizes with peer servers and schedule the next synchronization.
7. Extracts the timestamps from the data message and synchronizes using the timestamps.
8. Schedules the next synchronization time by adding the value of *nextPoll* seconds to the current time. At the next synchronization, goes to step 2.

Note: Application developers do not have to perform these steps; DTS performs these steps internally during synchronization with an external time-provider.

Running the Time-Provider Process

Both the TP process and the DTS daemon must run on the same system. The TP process must be started up under the login context of the machine's principal, which has root privileges. The DTS daemon and the TP process are started independently. However, before starting the TP process, ensure that **dced** is running on the system. If it is not running, start it. The TP process can always exit without affecting the DTS daemon. DTS dynamically reestablishes communications with the TP process when it creates binding handles.

Sources of Additional Information

Refer to the following for additional information:

- See **/examples/dts** for examples of time-provider programs that you can use with several different types of external time-provider devices.
- See the *OSF DCE Administration Guide—Core Components* for commercial sources of external time-providers.
- See the *OSF DCE Application Development Reference* for reference pages describing the RPC API and DTS API routines.

Chapter 22. DTS API Routines Programming Example

This chapter contains a C programming example showing a practical application of the DTS API programming routines. The program performs the following actions:

- Prompts the user to enter two sets of time coordinates corresponding to the timestamps of two “events.”
- Stores those coordinates in a **tm** structure.
- Converts the **tm** structure to a **utc** structure.
- Prints out the **utc** structure in ISO text format.
- Determines which event occurred first.
- Determines if Event 1 may have caused Event 2 by comparing the intervals.

```
#include time.h    /* time data structures       */
#include dce/utc.h  /* utc structure definitions     */

void ReadTime();
void PrintTime();

/* This program requests user input about events, then prints
 * out information about those events.
 */
main()
{
  struct utc event1,event2;
  enum utc_cmptype relation;

  /* Read in the two events.   */
  ReadTime(&event1);
  ReadTime(&event2);

  /* Print out the two events.   */
  printf("The first event is : ");
  PrintTime(&event1);
  printf("\nThe second event is : ");
  PrintTime(&event2);
  printf("\n");

  /* Determine which event occurred first.   */
  if (utc_cmpmidtime(&relation,&event1,&event2))
    exit(1);
  switch(relation)
  {
    case utc_lessThan:
    printf("comparing midpoints: Event1 < Event2\n");
    break;
    case utc_greaterThan:
    printf("comparing midpoints: Event1 > Event2\n");
    break;
    case utc_equalTo:
    printf("comparing midpoints: Event1 == Event2\n");
    break;
    default:
    exit(1);
    break;
  }

  /* Could Event 1 have caused Event 2? Compare the
   * intervals.
   */
  if (utc_cmpintervaltime(&relation,&event1,&event2))
    exit(1);

  switch(relation)
  {
    case utc_lessThan:
    printf("comparing intervals: Event1 < Event2\n");
    break;
    case utc_greaterThan:
    printf("comparing intervals: Event1 > Event2\n");
    break;
    case utc_equalTo:
    printf("comparing intervals: Event1 == Event2\n");
    break;
    case utc_indeterminate:
    printf("comparing intervals: Event1 ? Event2\n");
    default:
    exit(1);
    break;
  }
}
/* Print out a utc structure in ISO text format. */
void PrintTime(utcTime)
struct utc *utcTime;
{
  char  string[50];

  /* Break up the time string.
   */
  if (utc_ascgmtime(string,   /* Out: Converted time */
           50,     /* In: String length  */
           utcTime))  /* In: Time to convert */
    exit(1);
  printf("%s\n",string);
}

/* Prompt the user to enter time coordinates. Store the
 * coordinates in a tm structure and then convert the tm
 * structure to a utc structure.
 */
void ReadTime(utcTime)
struct utc *utcTime;
{
struct tm tmTime,tmInacc;
  (void)memset((void *)&tmTime, 0, sizeof(tmTime));
  (void)memset((void *)&tmInacc, 0, sizeof(tmInacc));
  (void)printf("Year? ");
  (void)scanf("%d",&tmTime.tm_year);
  tmTime.tm_year -= 1900;
  (void)printf("Month? ");
  (void)scanf("%d",&tmTime.tm_mon);
  tmTime.tm_mon -= 1;
  (void)printf("Day? ");
  (void)scanf("%d",&tmTime.tm_mday);
  (void)printf("Hour? ");
  (void)scanf("%d",&tmTime.tm_hour);
  (void)printf("Minute? ");
  (void)scanf("%d",&tmTime.tm_min);
  (void)printf("Inacc Secs? ");
  (void)scanf("%d",&tmInacc.tm_sec);
  if (utc_mkanytime(utcTime,
           &tmTime,
           (long)0,
           &tmInacc,
           (long)0,
           (long)0))
    exit(1);
}
```

Part 5. DCE Security Service

Chapter 23. Overview of Security

This chapter provides a brief overview of the two security services available in DCE:

- DCE Security Service
- Generic Security Services (GSS)

Refer to the *OSF DCE Application Development Reference* for detailed information on the Application Program Interfaces (APIs) discussed in the security chapters of this guide.

Purpose and Organization of the Security Chapters

This part of the guide explains the major features of DCE security so that you can decide what, if anything, you need to do to ensure that your DCE application is sufficiently secure. A lot of security is built into DCE, so in many cases you will need to do nothing, or very little, to secure your DCE application. Furthermore, you do not need to understand all of the details of the DCE security services in order to use them effectively.

Following the overview of the DCE Security Service in this chapter are two chapters that contain conceptual discussions of authentication and authorization. The remaining chapters in this part of the guide discuss the DCE Security Service APIs—registry, login, extended registry attribute (ERA), extended privilege attribute (EPA), key management, access control list (ACL), password management, and ID map—and GSS credentials.

About Authenticated RPC

Perhaps the most important security facility is the authenticated remote procedure call (RPC) facility. Authenticated RPC enables distributed applications to participate in authenticated network communications. Applications using the authenticated RPC routines may select the authentication protocol and the authorization protocol to be used, and set various protocol-independent protection levels for communicating with remote entities (users, servers, and computers).

The use of authenticated RPC is explained in “Chapter 13. Basic RPC Routine Usage” on page 183 and “Chapter 14. RPC and Other DCE Components” on page 195. “Chapter 14. RPC and Other DCE Components” on page 195 contains information about a number of RPC routines that relate directly to security issues, such as **rpc_binding_set_auth_info()**.

These security chapters, however, contains conceptual information that is useful for understanding the authentication and authorization protocols that authenticated RPC routines use; for this information, we recommend that you read “Chapter 24. Authentication” on page 493 and “Chapter 25. Authorization” on page 523, as well as this one.

About the GSSAPI

The GSS provides an alternate way of providing DCE security to distributed applications that handle network communications by themselves. With GSSAPI, you can include established applications in DCE and ensure the security and integrity of the applications and their data. In peer-to-peer communications, the application that establishes the secure connection is the context initiator or simply initiator. The context initiator is like a DCE RPC client. The application that accepts the secure connection is the context acceptor or simply acceptor. The context acceptor is like a DCE RPC server.

The GSS available with DCE includes two sets of routines:

- Standard GSSAPI routines, which are defined in the Internet RFC 1509 “Generic Security Service API: C-bindings.” These routines have the prefix **gss_**.
- OSF DCE extensions to the GSSAPI routines. These are additional routines that enable an application to use DCE security services. These routines have the prefix **gssdce_**.

The chapters that follow provide information about how the GSSAPI routines use the authentication and authorization protocols. “Chapter 26. GSSAPI Credentials” on page 533 provides information about GSS credentials, which are used to establish an application’s identity in DCE.

UNIX System Security and DCE Security

UNIX system security mostly presumes that a computer’s backplane can be trusted because computing operations are assumed to be local, and because the computer itself can be physically secured. In a distributed environment, the logical equivalent of the single system’s backplane is the network itself. Network computing means distributed, rather than localized, computing operations and, in the case of an open network (which DCE assumes), little of the network is physically secure. Thus, the nature of distributed systems poses special security risks, in addition to those posed by nondistributed systems. Unlike UNIX system security, DCE security is designed specifically to address those risks.

These considerations notwithstanding, network security is ultimately dependent on the security features that are local to the individual computers in the network and, what is more important, the manner in which those features are used and administered. Since any compromise to the local security of a computer in the distributed environment may introduce opportunities for compromising network security, DCE security does not diminish the importance of local security. In fact, the relative importance of local system security is greater in the distributed environment because the consequences of a local security breach may not be local. Finally, while DCE security does nothing to enhance local security, neither does it introduce any new avenues for compromising local security.

In the discussions in this guide, we assume you are familiar with the authentication and authorization features that UNIX systems provide: **/etc/passwd** and**/etc/group** file processing, routines that return or change file attributes, routines that return or change real or effective user IDs (UIDs) and group IDs (GIDs), and data encryption and decryption.

What Authentication and Authorization Mean

There are two questions that DCE security can answer for a principal about another principal with which it might want to communicate:

- Is this principal really who it says it is?
- Does it have the right to do what it wants to do?

Depending on the answers to these questions, a security-sensitive principal takes different courses of action with respect to a principal with which it is communicating.

To authenticate a principal means to verify that the principal is representing its true identity. To authorize a principal means to grant permission for the principal to perform an operation. While distinct, the concepts of authentication and authorization are also intertwined. For one thing, a principal's authorization is explicitly linked to its identity. For another, there is the possibility that authorization data concerning an authenticated principal can be falsified, which raises the additional question, "Should the authorization data concerning this principal be believed?" To this question also, DCE security can provide an answer to a principal for which this issue is a concern.

The discussions of authenticated RPC refer to the specific mechanisms by which authentication and authorization are performed as authentication and authorization protocols. Authenticated RPC supports at least one of each. However, RPC documentation refers to authentication and authorization protocols as services. The security chapters use the term protocol instead of service in this context to prevent confusion between the protocol-independent DCE authentication and authorization services and the various authentication and authorization protocols that those services support.

The GSSAPI combines authentication and authorization under a single security concept called a mechanism. The security mechanism provides applications a choice of either Kerberos security or Privilege Attribute Certificate (PAC) authorization under DCE security.

Authentication, Authorization, and Data Protection in Brief

When one principal talks to another in a distributed computing environment, there is a risk that communications between the two will provide a means for compromising the security of one or the other. For example, a client may attack a server, or a server may set a trap for clients. An attack is most likely to succeed if the malevolent principal can convince its victim that it is something other than what it really is (an attacker), and/or that it possesses authorization that it does not really have. A counterfeit identity and/or authorization data grants an attacker access that it presumably would not otherwise have, and so provides an opportunity for the attacker to do damage.

One way an attacker might obtain counterfeit credentials is to intercept network transmissions between a client and a server, and then attempt to decipher (and perhaps modify) the transmitted data. If the attacker is able to intercept *and decipher* a principal's authentication or authorization information, it can later use this data to masquerade as an authentic principal with proper authorization.

DCE security protects against these kinds of attacks. It contains features that enable principals to

- Detect whether data they receive has been modified in transit
- Be certain that an attacker will be unable to decipher any authentication and authorization data it may succeed in intercepting

DCE security gives DCE principals confidence that the identity and authorization of principals they communicate with are authentic.

Figure 65 on page 487 is an extremely condensed and highly stylized representation of the essentials of DCE security in terms of the DCE shared-secret authentication protocol and the DCE authorization protocol. Unless we note otherwise, assume that discussions in the security chapters of this guide refer to these two protocols, used in conjunction with one another.

The following is a description of the events depicted in the illustration:

1. Principal A (which could be an attacker masquerading as Principal A) requests authentication of its identity from the authentication service. This request is encrypted using several keys, one of which is a key derived from the password supplied by Principal A. A copy of Principal A’s key also exists in the registry database, having been stored there when the principal’s account was created (or when the password was changed). It is thus available to the authentication service.

   The authentication service then obtains the registry’s copy of Principal A’s key and uses it to decrypt Principal A’s authentication request. If the decryption succeeds, the keys are the same; Principal A is therefore authenticated and the authentication service replies with information that enables Principal A to ask the privilege service to authenticate its privilege attributes. (Privilege attributes are data used in making authorization decisions; they consist of the principal’s name and group memberships.) If Principal A fails to get authenticated privilege attributes (also referred to as credentials), it may simply assert its privilege attributes to Principal B.
2. Principal A now makes a request to Principal B to perform some operation that requires the **c** permission to object **d**, and presents its certified privilege attributes. Principal B may grant or deny **c** access to **d** after examining the ACL that protects object **d**. (An ACL associates the privilege attributes of principals with permissions to an object.) If **c** is one of the permissions listed in the ACL granted to Principal A, then Principal A is allowed to perform the operation; if the **c** permission is not granted, A is denied access.

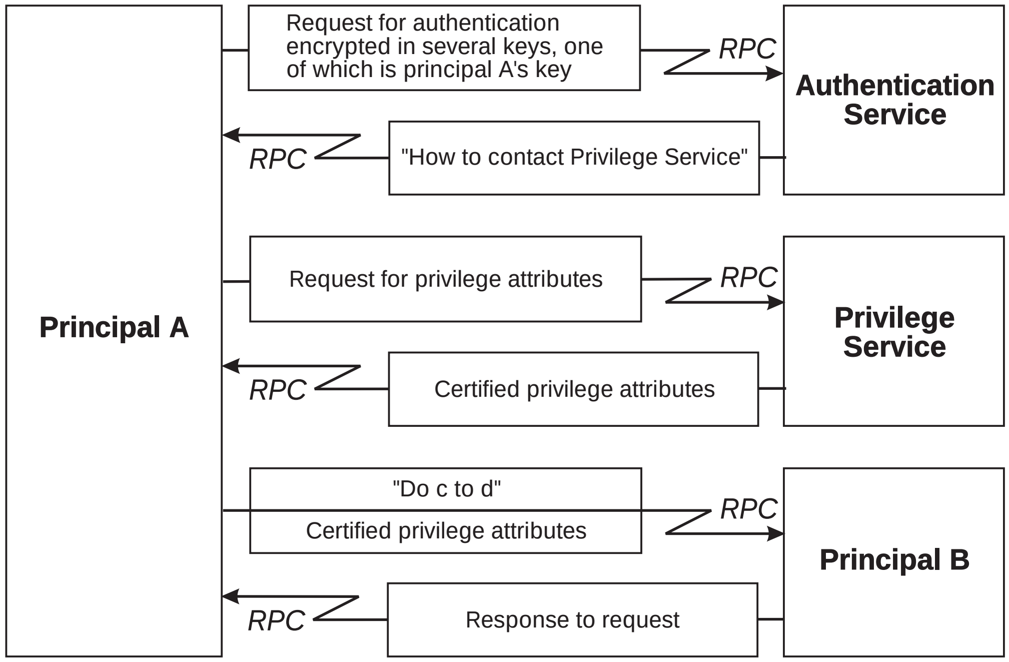

Principal A
Request for authentication encrypted in several keys, one of which is principal A's key
RPC
Authentication Service
"How to contact Privilege Service"
RPC
Request for privilege attributes
RPC
Privilege Service
Certified privilege attributes
RPC
"Do c to d"
Certified privilege attributes
RPC
Principal B
Response to request
RPC

Figure 65. Shared-Secret Authentication and DCE Authorization in Brief

Had the authentication service been unable to decrypt the principal's authentication request, the principal would have been unauthenticated and, as a consequence, unable to acquire certified privilege attributes from the privilege service. In that case, Principal A might have simply asserted its privilege attributes to B; that is, claimed them for itself, without the benefit of having the privilege service certify this data as being genuine. Had Principal A then presented asserted privilege attributes to Principal B, then B might have denied the requested permission or granted it, depending on whether B grants permissions to unauthenticated principals, and whether **c** is among the permissions that B grants to such principals.

If Principals A and B are especially sensitive to security concerns, they may request that transmitted data be checked for integrity to establish whether it has been modified in transit, and possibly also encrypted to ensure that the data is unintelligible to any party other than Principals A and B.

Summary of DCE Security Services and Facilities

The DCE Security Service consists of services and facilities. The security services are

- The registry service, which maintains a database of principals, groups, organizations, accounts, and administrative policies.
- The authentication service, which verifies the identity of a principal and issues tickets that the principal uses to access remote services. (A ticket is data about a principal that is presented to the entity providing the service.)
- The privilege service, which certifies a principal's privilege attributes (that is, its name and group memberships, which are represented as UUIDs).

The three security services are implemented in a single daemon, the security server.

The DCE Security Service facilities are

- The login facility, which enables a principal to establish its network identity.
- The ERA facility, which extends the registry database to maintain attribute types and instances.
- The EPA facility, which provides access to the information in extended privilege attribute certificates (EPACs)
- The ACL facility, which enables a principal's access to an object to be determined by a comparison of the principal's privilege attributes to the object's permissions.
- The key management facility, which enables noninteractive principals (most frequently, servers) to manage their secret keys.
- The ID map facility, which maps cell-relative principal names to global principal names, and global principal names to cell-relative principal names. This facility is used in connection with the transmission of information about principals that are members of different DCE cells.
- The password management facility, which enables principal's passwords to be generated, and to be subjected to strength-checks beyond those defined in DCE standard policy.

For UNIX system compatibility with DCE, the DCE Security Service also provides implementations of UNIX system C library interfaces to the **/etc/passwd** and **/etc/group** files.

Interfaces to the Security Server

Following are the user interfaces to the security server itself (see the *OSF DCE Administration Guide—Core Components* and the *OSF DCE Administration Commands Reference*):

- **secd**

  The security daemon (a replicated server)
- **sec_create_db**

  Creates the security databases
- **sec_admin**

  Administers instances of the security daemon
- **sec_salvage_db**

  Converts the security database from one version of DCE to another

  Salvages a corrupted security database
- The security validation service of **dced**

  Enables clients of the security server to communicate with it

All other interfaces to the security server are more precisely characterized as interfaces to its three services: registry, authentication, and privilege.

Registry Service Interfaces

User interfaces to the registry service are described in the *OSF DCE Administration Guide—Core Components* and the *OSF DCE Administration Commands Reference*. Following is a summary of them:

- **rgy_edit**

  Edits registry database entries
- **passwd_import**

Creates registry database entries from UNIX system **/etc/passwd** and **/etc/group** files

- **passwd_export**

  Creates local registry information that corresponds to network registry database entries
- **chpass**

  Changes a user’s password in a registry database entry

Authentication Service Interfaces

Following is a summary of the user interfaces to the authentication service when the default authentication protocol is in effect (the default protocol is DCE shared-secret, which is based on the Kerberos Version 5 network authentication system).

- **kinit**

  Obtains a login session’s ticket(s) to remote services (the **login** and **su** tools also perform this service)
- **klist**

  Lists a login session’s tickets to remote services
- **kdestroy**

  Destroys a login session’s tickets to remote services

There are two security APIs that distributed applications are most likely to call to use the authentication service:

- Authenticated RPC facility
- GSSAPI

Although an application that uses GSSAPI may not make explicit calls to RPC routines, the GSSAPI implementation itself uses DCE RPC to communicate with the DCE registry.

Privilege Service Interfaces

There are no user interfaces or APIs to the privilege service. The login facility and authenticated RPC or GSSAPI encapsulate interactions between a principal and the privilege service.

Interfaces to the Login Facility

User interfaces to the login facility consist of the following tools:

- **dce_login**

  Enables an interactive principal to log into DCE, but does not change the principal’s local identity
- **login**

  Enables an interactive principal to log in
- **su**

  Enables a logged-in interactive principal to assume a different principal identity

The API to the login facility consists of calls that are prefixed with **sec_login_**. This API enables application processes to assume their network identities. Network login and system login programs are examples of applications that call this API.

Interfaces to the Extended Registry Attribute Facility

The user interface to the ERA facility consists of DCE control program (**dcecp**) commands that allow users to modify the registry schema to create and maintain attribute types and to create and maintain instances of those types.

The API to the ERA facility consists of calls that are prefixed with **sec_rgy_attr_**.

Interfaces to the Extended Privilege Attribute Facility

There are no user interfaces to the EPA facility. The API to this facility consists of calls that are prefixed with **sec_cred_**. These routines extract data from EPACs.

Interfaces to the Key Management Facility

For a distributed application, it may be important for a server to have a network identity that is distinct from the principal identity it inherits from the user who invokes it or the host on which it runs. The key management facility provides features that enable noninteractive principals to manage their secret keys.

The user interface to the key management facility consist of a few **rgy_edit** subcommands that enable an administrator to maintain a key table. A remote interface allows users and administrators to maintain key tables on remote machines through the **dcecp keytab** verbs. A subset of local operations is also available though this interface. These subcommands call the key management API, which consists of several calls with the prefix **sec_key_**.

Interfaces to the ID Map Facility

There are no user interfaces to the ID map facility. The API to this facility consists of calls that are prefixed wht **sec_id_**. These routines map a global principal or group name into a cell name and a cell-relative principal or group name, and generate a global principal or group name from a cell name and a cell-relative principal or group name. This API also converts between the internal (UUID) representation of a name and the human-readable string.

Interfaces to the Access Control List Facility

The only user interface to the ACL facility is the **dcecp** ACL object **acl_edit**. This tool edits an object's ACL, the entries of which specify the permissions to the object that may be granted to principals possessing specified privilege attributes.

The ACL API consists of routines that are prefixed with **sec_acl_**. This is the same API that **acl_edit** calls, so an ACL editor or browser that is intended to replace **acl_edit** would call this API. A different case is that of an application server that needs to store and retrieve application-specific, access-control information for its clients. Such an application needs to implement its own ACL manager by using the DCE ACL library. (Refer to "Chapter 32. The Access Control List APIs" on page 591 for more information on ACL managers).

DCE Implementations of UNIX System Program Interfaces

DCE security provides implementations of UNIX system C library interfaces related to security. These are **getpwent()** and the related program interfaces to the

/etc/passwd file, and **getgrent()** and the related program interfaces to the **/etc/group** file. Applications that bind with **libdce.a** are bound with the DCE security implementations of these interfaces.

Interfaces to the Password Management Facility

The user interface to the password management facility is provided by subcommands to the **rgy_edit** and **dcecp** commands. These subcommands enforce password management policy for principals and enable them to request generated passwords. See the **rgy_edit(8sec)** and **dcecp(8dce)** reference pages and the *OSF DCE Administration Guide—Core Components* for information on using these commands to create and change principal passwords.

The API to the password management facility consists of routines that are prefixed with **sec_pwd_mgmt_**. See the appropriate reference pages and “Chapter 36. The Password Management API” on page 621 for information on these routines.

Relationships Between the DCE Security Service and DCE Applications

Figure 66 is a schematic illustration of the relationships among the interfaces to the DCE Security Service, and the relationship of security interfaces to DCE applications.

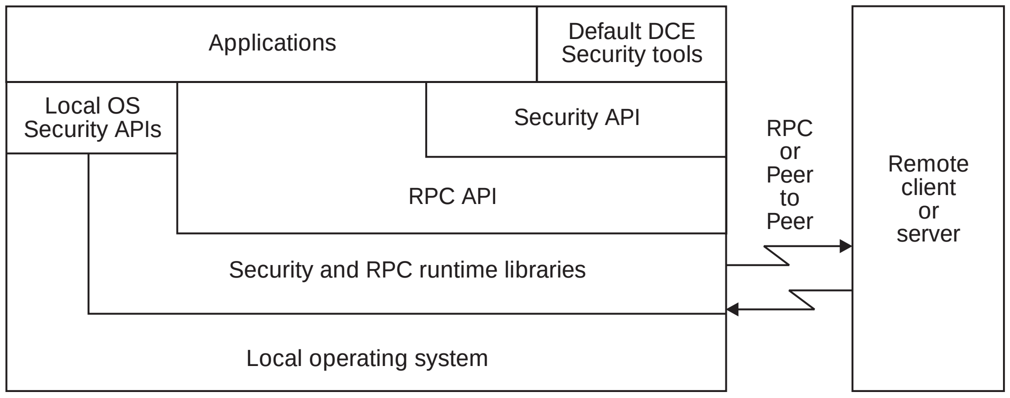


Figure 66. DCE Security and the DCE Application Environment

DTS, the Cell Namespace, and Security

The following subsections discuss the dependencies of DCE security on the Distributed Time Service (DTS), and the relationship between the security namespace and the Cell Directory Service (CDS) namespace. For information about how DCE components such as CDS use features of DCE security, refer to the documentation on the component of interest (for example, the section of the *OSF DCE Administration Guide—Core Components* on CDS).

DTS and Security

The DCE Security Service depends on a relatively close synchronization of network clocks, a service provided by DTS. When network clocks become too skewed, unexpired tickets to services may be regarded as invalid, and/or expired tickets considered valid. Excessive skewing can inconvenience users and introduce opportunities for security breaches; in the latter case, administrative intervention is required.

The Cell Namespace and the Security Namespace

The registry database maintains three security namespaces: the principal, group, and organization (PGO) namespaces. These namespaces are distinct from the cell namespace maintained by CDS. Security names take the following form:

```
/.../ cell_name/ pgo_name
```

CDS names take the following form:

```
/.../ cell_name/ pathname/ object_name
```

Since the security namespace is rooted in the CDS namespace, security names have equivalent CDS names. Thus, for example, an entry for a principal in the registry database has the first of the following forms in the security namespace and the second of the following forms in the CDS namespace:

```
/.../ cell_name/ principal_name
  /.../cell_name/ security_mount_point/ principal/ principal_name
```

Note: The security mount point (*security_mount_point* as shown in the preceding syntax) is determined when DCE is configured. Therefore, the name may differ at individual sites.

There is no ambiguity about the security namespace to which a name refers because security names are always used in contexts that identify the namespace in question. For example, logging into DCE requires a principal name to be supplied.

However, an ACL is an object that is referenced not directly, but by the name of the object it protects. Since protected objects are not always security objects (and therefore may be registered *only* in the CDS namespace), ACL management interfaces always take CDS names rather than security names as input, whether or not it is the ACL of a security object (such as a registry database entry) that is being read or modified.

Chapter 24. Authentication

This chapter describes the authentication process of users and applications, as well as of principals in other cells.

Note: The authenticated RPC facility may also be referred to as the *protected* RPC facility, as it involves services beyond authentication.

Background Concepts

The following concepts, as they relate to this chapter, are described within this section:

- Principals, which are the subjects of authentication
- The shared-secret authentication protocol, which is the mechanism by which authentication is effected when applications specify this protocol via the authenticated RPC facility
- Cells, which are the environment where authentication takes place
- Protection levels, which are the various degrees to which transmitted application-level data may be protected
- Data encryption/decryption (cryptographic) algorithms, which are the mechanisms that the security server and client and server runtimes use to encrypt and decrypt data exchanged between principals

Principals

For the purposes of this discussion, the term principal may be precisely defined as an entity that is capable of believing it can communicate securely with another entity. In DCE security, principals are represented as entries in the registry database. DCE principals include the following:

- Users, who are also referred to as interactive principals
- Instances of DCE (system-level) servers
- Instances of application-level servers
- Computers (hosts) in a DCE cell
- Key distribution service (KDS) surrogates (these are used for cross-cell authentication; see “Intercell Authentication” on page 519)

The DCE security server itself comprises three principals that correspond to the three services that it provides: KDS, registry service, and privilege service. The KDS in turn provides two subservices: the authentication (sub)service and the ticket-granting (sub)service (TGS).

Note: As used in the literature, the term authentication service is sometimes ambiguous. This name may be, in places, associated with at least three distinct entities: the authentication (sub)service of the KDS, the KDS itself (comprising its authentication and ticket-granting subservices), and the entire DCE Security Service (comprising the KDS, the registry service, and the privilege service).

These three servers (KDS, registry service, and privilege service) comprise the main part of the DCE network trusted computing base. The KDS, registry service, and privilege service servers are commonly all implemented in a single process called the security server or security daemon.

The Shared-Secret Authentication Protocol

The registry service maintains a database, which contains an entry representing every principal, identifying the principal by its name and a secret key bound to it. It is this binding of the principal identity to a secret key shared with the registry that is at the root of the DCE shared-secret authentication protocols, as will be seen in this chapter. In the case of an interactive principal, the secret key is derived from the user's password (at login time). In order to establish its identity as a principal, a noninteractive principal, such as a server or computer, must store its secret key in a data file or hardware device, or rely on a system administrator to enter it. The secret keys of servers are considered to be stronger than those of users/clients, because they are truly random (as opposed to being derived from a password, which greatly restricts their randomness).

DCE shared-secret authentication implements an extended version of the Kerberos Version 5 system as its authentication protocol. Namely, the part of the DCE security server that corresponds to Kerberos is the KDS. The other parts (registry service and privilege service) do not occur in Kerberos. The Kerberos system was originally developed at the Massachusetts Institute of Technology as part of Project Athena, and provides a trustworthy, shared-secret authentication system. The walkthrough of the authentication protocol in this chapter describes the protocol in general terms.

Note: The KDS is an exceptional principal in that it does not share its key with any other principal. KDS surrogates (see "Intercell Authentication" on page 519) are also exceptional in that they are not autonomous participants in authenticated communications, as other kinds of principals are.

In the theory of shared-secret authentication all principals are initially considered to be untrusted, except for those in the trusted computing base itself (KDS, registry service, privilege service). A security-sensitive application must make use of the trusted computing base to convince itself of the level of trust it may place in all other principals. How that is done is the subject of this chapter.

Cells and Realms

The cell is the basic unit of configuration and administration in DCE. In terms of security, a cell is the set of principals that share a secret key with an instance of the registry service. Therefore, each instance of a security server (together with its replicas) defines a separate cell.

From the perspective of security only, a cell is sometimes also known as a realm or security domain. (The term realm is often used in Kerberos documentation, and so may be more familiar to some readers than is the term *cell*.) A security cell is always configured to coincide with a corresponding CDS cell, and perhaps Distributed File System (DFS) cell as well. DCE documentation always refers to such a collective configuration of services as a cell.

Protection Levels

Protection levels specify how much of the information in network messages exchanged by principals is encrypted. As a rule, the higher the protection level, the greater the negative impact on performance. An application can set a protection level by using either authenticated RPC or GSSAPI.

Authenticated RPC and Protection Levels

The authenticated RPC facility provides several levels of protection so that applications can control tradeoffs between security and performance. Following is a summary of some of the protection levels that an application using authenticated RPC may specify:

- Connect level

  Performs authentication only when a client and server establish a relationship (or connection)
- Call level

  Attaches a verifier to each client call and server response that protects the system-level metadata of every remote call (but not the application-level data)
- Packet-integrity level

  In addition to protecting metadata, ensures the integrity of the application-level data (RPC and return parameters) transferred between two principals, that is, that none of it has been modified in transit
- Packet-privacy level

  In addition to protecting metadata and integrity, encrypts all application-level data, thus guaranteeing its confidentiality

Refer to the discussion of authenticated RPC in “Chapter 13. Basic RPC Routine Usage” on page 183 and “Chapter 14. RPC and Other DCE Components” on page 195 for complete information about protection levels.

GSSAPI and Protection Levels

Unlike authenticated RPC, where the client chooses a protection level that is then applied automatically to all data transferred in either direction, applications that use GSSAPI must explicitly protect data on a message-by-message basis. This allows an application the option of protecting only particularly sensitive messages, and avoids the overhead of security processing for other messages. (That is possible with RPC too, of course, provided that the programmer is willing to specify security attributes on an RPC call-by-call basis.)

GSSAPI offers two distinct types of protection through the **gss_sign()**/ **gss_verify()** routines and the **gss_seal()**/**gss_unseal()** routines, as follows:

- The **gss_sign()** routine creates a token containing an encrypted signature to protect the integrity of the message data. The token contains only the signature (not the message data). The application must send both the token and the message to which it applies to the peer application for verification. The receiving application calls the **gss_verify()** routine to check the signature.
- The **gss_seal( )** routine creates a token containing both an encrypted signature and the message data, and may optionally encrypt the message data. Only the token need be sent to the peer application, which processes it by using the **gss_unseal()** routine to verify the signature and extract the message data.

Three distinct signature algorithms are supported by the per-message protection routines. An algorithm may be requested by providing one of several constants to the **qop_request** parameter (**qop** stands for quality of protection) of either the **gss_sign()** or the **gss_seal()** routine. The constants are as follows:

GSSDCE_C_QOP_DES_MAC
Conventional DES MAC. Slow but well understood.

GSSDCE_C_QOP_DES_MD5
DES MAC of an MD5 (Message Digest #5) signature. Faster than DES MAC.

GSSDCE_C_QOP_MD5
MD5 signature. Fastest supported signature algorithm. The default.

Data Encryption Mechanisms

Authentication protocols assume the availability of a data encryption mechanism, parameterized by a so-called crypto-variable or key. In fact, it is the knowledge of such a key that is the concrete manifestation of the abstract notion of authentication. One mechanism that is frequently used is the Data Encryption Standard (DES), though the DCE security architecture supports other cryptographic algorithms. Your version of DCE security may use DES for data privacy or for principal authentication and data-integrity checking; or it may use another encryption mechanism, or no encryption at all. Consult the documentation supplied by your DCE vendor for specific information.

A Walkthrough of Shared-Secret Authentication Protocols

This section walks you through the following topics:

- Authentication of the user during login, in “Authenticating a User” on page 497.
- Authentication of applications, in “Authentication Using Authenticated RPC” on page 512.

The walkthrough is seen primarily from the user and the associated application-client side. The illustrations in this chapter show only a high-level view (not low-level details) of what happens when a user logs in and runs an authenticated application; they are intended only to provide a general understanding of the protocol. (See the *Security Volume* of the *Application Environment Specification/Distributed Computing* for full details.)

In these figures, fill patterns represent encryption key values and encrypted data. The key symbol within a box indicates that a key is being passed as data. The key symbol on a line indicates that encryption or decryption is taking place, depending on whether the resulting data is represented as encrypted or not. These conventions are shown in Figure 67 on page 497.

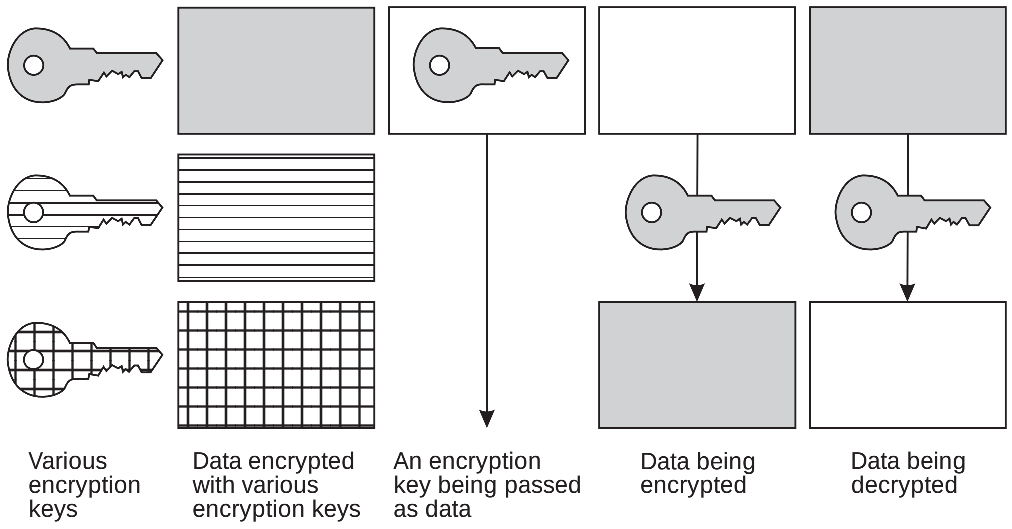

Various encryption keys
Data encrypted with various encryption keys
An encryption key being passed as data
Data being encrypted
Data being decrypted

Figure 67. Conventions Used in Authentication Walkthrough Illustrations

Note: All computer-to-computer communications initiated by DCE security are processed through the RPC mechanism, although the integration of security with client and server RPC runtimes are not illustrated or explained in any detail here.

Finally, note that to use shared-secret protocol, you do not need to understand how it works. It is described here so that application developers can determine whether it provides sufficient security for their needs. The discussion that follows is technical and detailed and may not be of interest to every reader.

Authenticating a User

This section explains how DCE security authenticates a user/client. DCE authentication basically consists of two successive procedures:

1. Acquisition by the security client of a ticket-granting ticket (TGT) for the user.
2. Acquisition by the security client of a privilege-ticket-granting ticket (PTGT) for the user.

These procedures are described in the following two subsections.

How the Client Obtains a TGT for the User

This section describes the acquisition, by the security client, of the user's TGT. It is the authentication service of the KDS that issues TGTs. Acquisition of the user's TGT is the first of the two parts of DCE user authentication. The other part is the acquisition of service tickets, which are issued by the TGS of the KDS.

Authentication protocols used by DCE security clients and servers to obtain TGTs for users, which is the first part of the user-authentication process, are:

- The *public key* protocol, which provides the highest level of security
- The *third-party* protocol, which is less secure than public key protocol
- The *timestamps* protocol, which is less secure than the third-party protocol

- The DCE Version 1.0 protocol, which is the least secure of the three and is provided solely to enable DCE Version 1.1 security servers to process requests from pre-DCE Version 1.1 clients

The protocol used by the security client when it makes a login request to the authentication service is determined as follows:

1. Pre-DCE Version 1.1 clients always use the DCE Version 1.0 protocol.
2. DCE Version 1.1 clients always use the third-party protocol, unless the host machine's session key, which the client uses to construct the request, is unavailable. It then uses the timestamps protocol.
3. DCE Version 1.2.2 clients always attempt to use the public key authentication protocol. If a client is unable to use the protocol, the client reverts to DCE Version 1.1 behavior.

The protocol used by the authentication service to respond to the client is determined by the following:

- The protocol used by the client making the login request
- The value of any **pre_auth_req** ERA attached to the requesting principal

The authentication service always attempts to reply by using the same protocol used by the client making the request, unless the value of the ERA forbids it to do so. (See the discussion of DCE Version 1.1 authentication in the *OSF DCE Administration Guide—Core Components* for more detailed information on how security clients and the authentication service determine which protocol to use.)

For a general discussion of the security aspects of these protocols, and of security administration and security ERAs, see the *OSF DCE Administration Guide—Core Components*. The following subsections explain how the three protocols operate.

The Public Key Authentication Protocol: Public key authentication protocol works via public and private key-pairs. A user's identity is proven to the DCE Key Distribution Center (KDC) through a signature in the user's private authentication key. The KDC verifies the request through the user's authentication public key, which must be contained in the DCE registry. If the request is verified, the KDC replies with a TGT that is first signed by the KDC using its own private authentication key, and then is encrypted by the KDC using the client's key encipherment public key, which must be stored in the DCE registry. Because the KDC does not know the user's private keys, a compromise of the KDC cannot reveal the private keys. Therefore, public key users will not have any identifying information exposed to an intruder. This method of public and private key pair usage constitutes the public key protocol.

The public key protocol invokes routines **sec_login_validate_identity()**, **sec_login_valid_and_cert_ident()**, and **sec_login_validate_first()** as follows:

1. The user logs in.
2. The client process sends a message to the KDC. The message consists of a timestamp and nonce signed by the client's private digital signature key. An optional certificate of the client can also be sent along.
3. The KDC checks the timestamp and signature of the client's message. If the information is valid, the KDC sends a reply key to the client. The reply consists of a message signed by the KDC's digital signature key and then encrypted by the client's key encipherment key. The reply key is for encrypting the encrypted portion of the Kerberos **KRB_AS_REP** message, which includes the symmetric

session key associated with the TGT. The session key used in association with the TGT is returned in the standard **EncKDCRepPart** field of the **KRB_AS_REP** message.

If the KDC is unable to authenticate the user's supplied public key data, the KDC returns an error indicating why the authentication failed and whether the user is required to use the public key authentication protocol. The KDC determines this from the **pre_auth_req** ERA attached to the user principal.

If the public key login attempt fails, the **sec_login** code instead falls back to the use of existing password-based authentication unless the KDC error information indicates that the principal is required to use public key login authentication. Preventing fall back is done by giving each principal a **pre_auth_req** ERA value of **PADATA_ENC_PUBLIC_KEY**.

Authentication information is transmitted as data types:

- **KRB5_PADATA_PUBKEY_REQ**
- **PADATA_ENC_PUBKEY_REP**

4. The client checks the signature on the reply to make sure it is from the KDC. The sesion key can be decrypted only by the legitimate client that possesses the private key needed to decrypt. The client then uses the TGT and associated session key.

Storage of the Private Key: Private key information is stored either in a local file or by the DCE private key storage server (PKSS). If the principal's `DCEPKPrivateKeyStorage` ERA value is not set, the login program assumes the private key is stored in a local file. If the principal's **DCEPKPrivateKeyStorage** ERA value is set, the login program obtains the private key from the private key storage mechanism associated with the UUID contained in the ERA. The currently supported storage mechanisms and their associated UUID's are the following:

- Local file — The UUID is **8687c5b8-b01a-11cf-b137-0800090a5254**.
- Private key storage server (PKSS) — The UUID is **72053e72-b01a-11cf-8bf5-0800090a5254**.
- Registry Database — The UUID is **adb48ed4-e94d-11cf-ab4b-08000919ebb5**. (This mechanism is supported for internal security server purposes only.)

The PKSS stores private keys in records that have the following information:

- The user's principal name.
- The user's public key.
- The key version of the user's public key (key v.n).
- The application domain. (Currently, private keys are used only in the context of a DCE login.)
- Key usage flags. (Currently, private keys are used only for authentication and for key encipherment.)
- Password hash value 2 (H2) derived from the user's password.
- The user's private key encrypted under the user's password hash value 1 (H1).

The PKSS cannot directly access the user's private key because it does not have the user's password H1 value. An ACL protects user records from unauthorized access, allowing access to only the **sec_admin** principal.

The following two descriptions depict the initial message exchange between the login client and the PKSS, and the second (final) exchange in which the PKSS returns the private key to the client.

Client Initiation of Private Key Acquisition from PKSS

The client DCE login program begins the process of key acquisition from the PKSS. Refer to Figure 68 on page 501 as you read the following steps.

1. The login client sends a message to the PKSS that consists of the following components:
   - The user's principal name.
   - The application domain.
   - Key usage flags.
   - The key version number (key v.n).
   - An exponentiated Diffie-Hellman value (S^c) used for establishing a Diffie-Hellman key.
   - An algorithm list (alg list), which is a list of secret key encryption algorithms supported by the client (currently only DES). The client and the PKSS use this algorithm with the Diffie-Hellman key and the session key.
2. Upon receipt of this message from the login client, the PKSS generates a Diffie-Hellman value of its own (S^s). Using this value along with the client's Diffie-Hellman value, the PKSS computes a Diffie-Hellman key.

   The PKSS determines whether it supports any of the algorithms listed in the client message. If so, it can communicate securely with the client and the PKSS selects one of the supported algorithms for use. (Currently, OSF DCE clients and servers support only DES.)
3. The PKSS generates a random session key and a nonce (N^s). The session key will be used to encrypt messages between the client and server. The PKSS encrypts the nonce (N^s) with the session key, then encrypts the session key with the user's password H2 value taken from the user record.
4. The PKSS computes a hash on the algorithm list provided by the client. It encrypts both the hashed algorithm list and the encrypted session key (see step 3) under the Diffie-Hellman key generated in step 2.
5. The PKSS composes and sends the client a message consisting of
   - The nonce (N^c) encrypted under the random session key.
   - The session key encrypted by the user's password H2 value and then encrypted under the Diffie-Hellman key.
   - The hashed algorithm list further encrypted under the Diffie-Hellman key.
   - The algorithm to be used for the session key. (The algorithm is chosen from the client's algorithm list.)
   - The PKSS-generated Diffie-Hellman value.
6. Upon receipt, the login client extracts the PKSS-generated Diffie-Hellman value and combines it with its own Diffie-Hellman value to obtain its copy of the Diffie-Hellman key.
7. The client uses the encryption algorithm specified by the PKSS, along with its Diffie-Hellman key to obtain the hashed algorithm list and the session key (still encrypted under the user's password H2 value).

   The login client computes a hash on its own algorithm list and compares it with the hashed algorithm list returned from the PKSS. The two lists must match. Otherwise, the client determines that the PKSS is invalid and returns an error.
8. The login client decrypts the session key by using H2 derived from the user's password. The client uses the session key to decrypt the PKSS-generated nonce (N^s). The session key will be used to authenticate the current session communications between the client and PKSS.

Note: A PKSS imposter would not know the user's password H2 value. The resulting session key would differ from the imposter's session key, preventing further communications between the client and the imposter.

9. The client composes a message encrypted under the session key. The message consists of
   - The PKSS-generated nonce (N^s)
   - A client-generated nonce (N^c)
   - An operation identifier that indicates private key acquisition

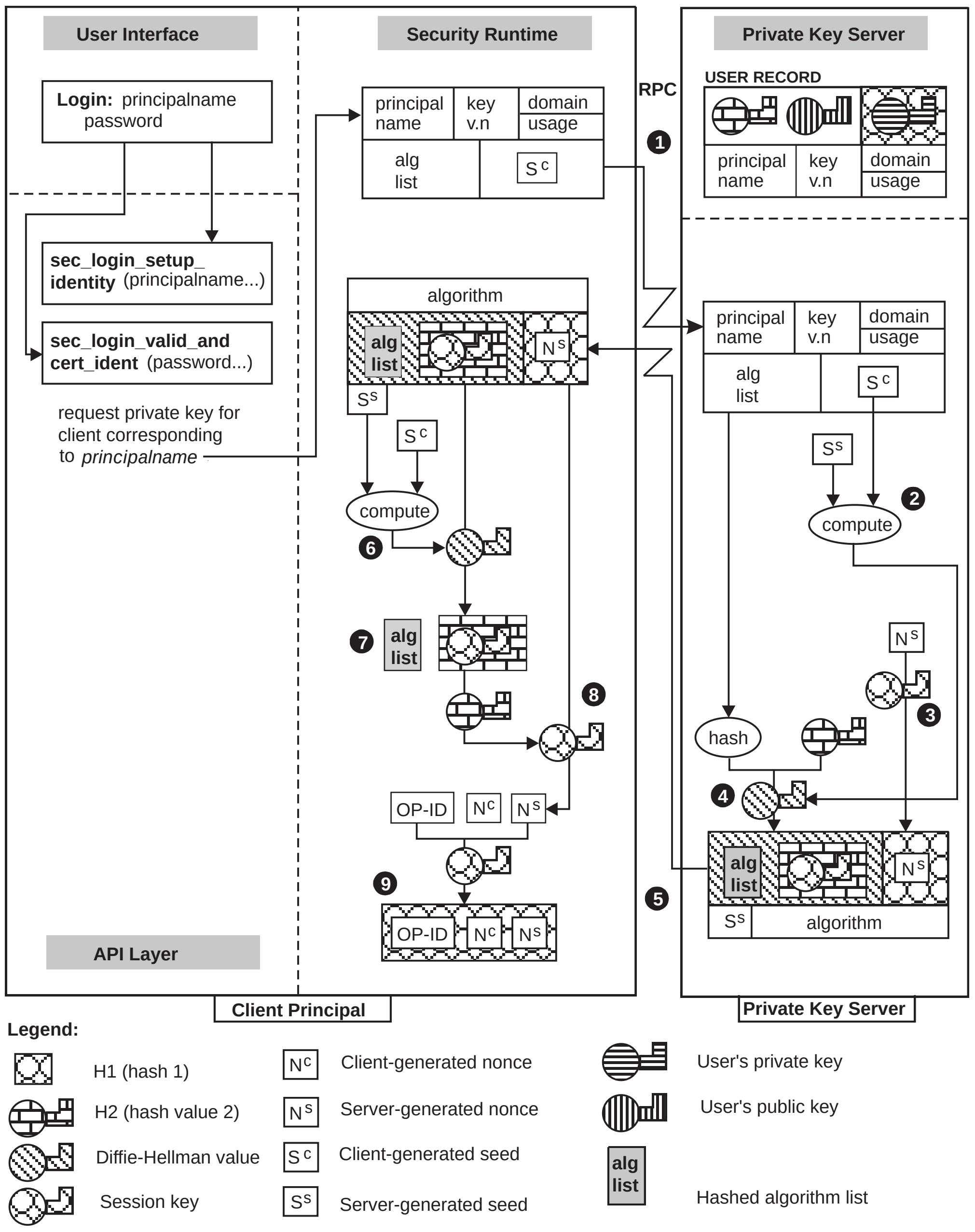


Figure 68. Client Initiation of Private Key Acquisition

Client Completion of Private Key Acquisition from PKSS

The client DCE login program completes the process of key acquisition from the PKSS. Refer to Figure 71 on page 508 as you read the following steps.

1. The client sends the PKSS the composed message encrypted under the session key (see Step 9 in the preceding discussion).
2. Upon receipt of the client message, the PKSS uses the session key to obtain the operation ID, the client-generated nonce (N^c), and the PKSS-generated nonce (N^s).

   The PKSS compares the client's copy of N^s with its original nonce (N^s). A match proves the client had knowledge of the user's secret password which was needed to obtain the session key. The client used the session key to obtain N^s.
3. The PKSS composes a message consisting of
   - The user's private key encrypted under the user's password H1 value
   - The user's public key
   - The key version number
   - The client-generated nonce (N^c)
4. The PKSS encrypts this message with the session key and sends it to the client.
5. Upon receipt, the client uses the session key to obtain the user's private key (encrypted under the user's password H1 value), the user's public key, the key version number, and the client-generated nonce (N^c).

   The client compares the PKSS's copy of N^c with its own nonce. A match proves the authenticity of the PKSS because only the true PKSS could have used the correct user password H2 value to properly encrypt the session key passed to the client in the first message.
6. The client uses its password H1 value to decrypt the private key. The client's security runtime program returns the authenticated private key to the calling routine.

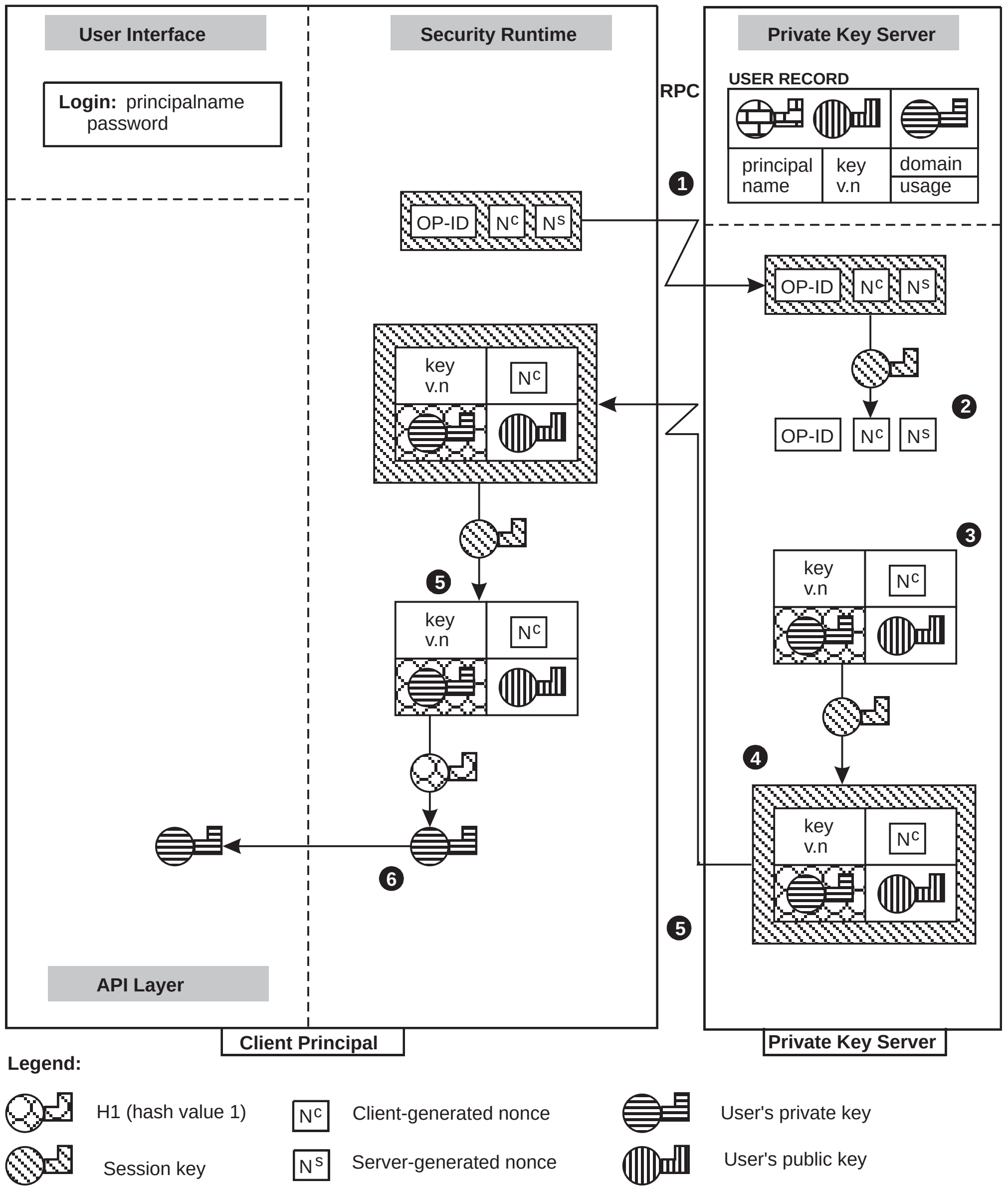


Figure 69. Client Acquisition of Private Key from PKSS

The Third-Party Authentication Protocol: The DCE Authentication Service can use the third-party authentication protocol to provide a user with a TGT. Refer to Figure 68 on page 501 as you read the following steps.

1. The user logs in, entering the correct user name. The login program invokes **sec_login_setup_identity()**, which takes the user's principal name as one of its arguments, and **sec_login_valid_and_cert_ident()**, which has the user's password as one of its arguments. The **sec_login_valid_and_cert_ident()** routine causes the security runtime to request a TGT from the authentication service of the KDS. (The client principal will later present the TGT to the TGS, to acquire service tickets to other servers.) The client's security runtime performs the following steps to construct the TGT request to the authentication service:
   a. It requests, from the **secval** service, a random key, say conversation key 1, which the client will later use to encrypt its request to the authentication service. Two copies of conversation key 1 are passed to the client: one

unencrypted and one encrypted in the machine session key (a copy of which is sealed inside the machine ticket-granting ticket, or MTGT). (In order to do this securely, the request to **secval** must be done over a secure local communications channel on the host machine.) It then concatenates the encrypted copy of conversation key 1 with the MTGT.

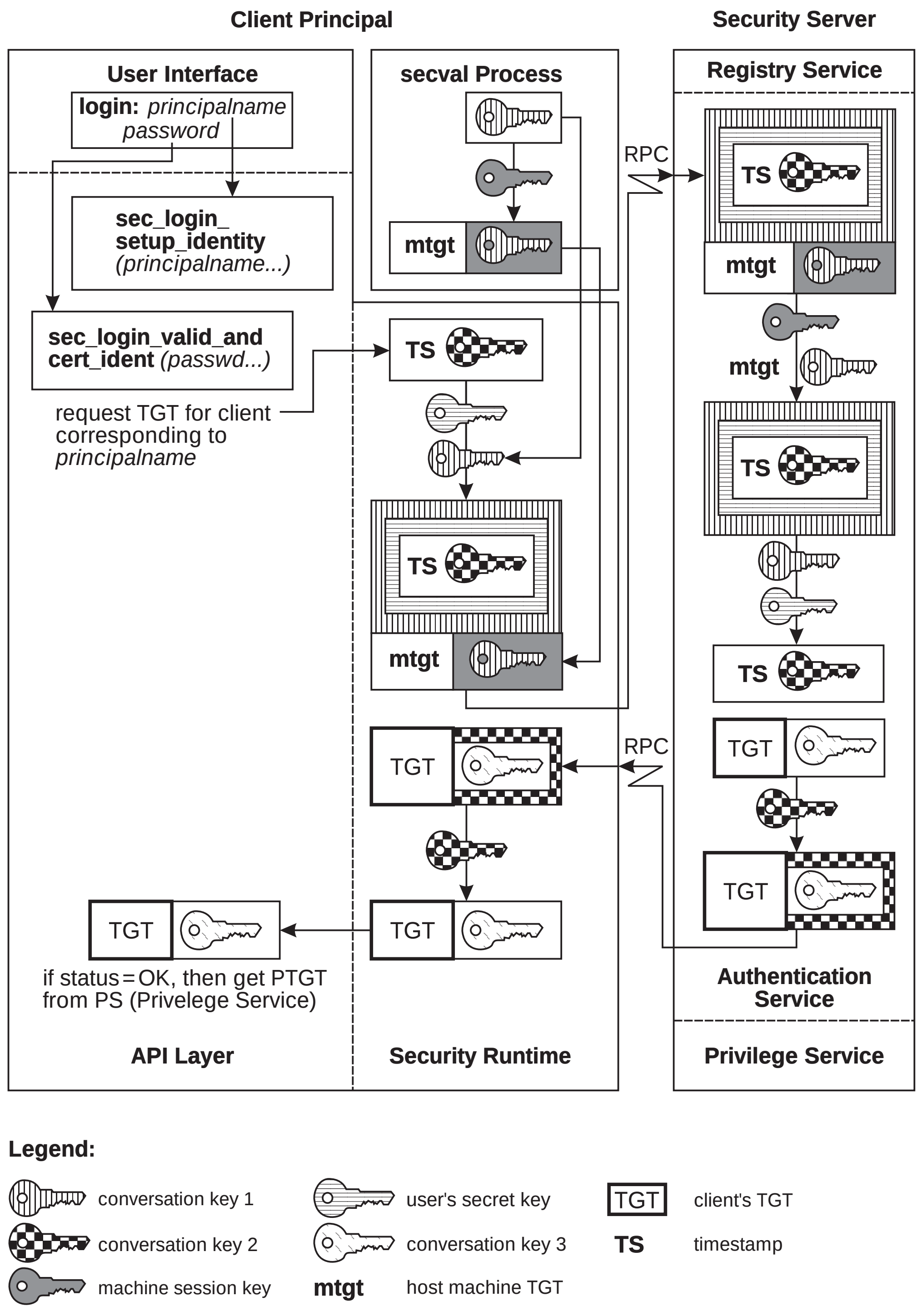


Figure 70. Client Acquires TGT Using Third-Party Protocol

   b. It generates another random key, conversation key 2, which the authentication service will later use to encrypt the TGT it returns to the client. It then concatenates it to a timestamp string.
   c. It derives, from the password input by the user, the user's secret key, a copy of which also exists in the registry service database. It then encrypts the timestamp/conversation key 2 twice: first by using the user's secret key, and then by using conversation key 1.
   d. Finally, it completes constructing the authentication service request message by concatenating the encrypted conversation key 1 (obtained from **secval** in Step 1a) with the doubly encrypted timestamp and conversation key 1.
2. The client's security runtime then forwards the constructed request to the authentication service of the KDS. (This corresponds to the first step of the DCE Version 1.0 protocol, described in "The Third-Party Authentication Protocol" on page 503.)
3. The authentication service receives the request and performs the following steps to verify the user and prepare the user's TGT:
   a. It decrypts the MTGT (by using the KDS's secret key), and obtains the machine session key from it. (This decryption is not shown pictorially in Figure 68 on page 501.)
   b. Using the machine session key, it decrypts the package containing conversation key 1.
   c. It obtains the user's secret key from the registry service and then decrypts the doubly encrypted package containing the timestamp and conversation key 2 by using the user's secret key and conversation key 1.

      If this decryption fails, the user's secret key that was used by the login program to encrypt the package differs from the one stored in the registry service, and therefore the password supplied to the login program by the user was incorrect. In this case, the user is not authenticated, and an error code is returned to the login program.

      If the decryption succeeds, and if the decrypted timestamp is within an allowable clock skew (5 minutes) of the current time, the user has been authenticated (that is, the user knows the correct principal password and this isn't a replay attack), and the authentication service proceeds with preparation of the user's TGT.
4. The authentication service then prepares the user's TGT, encrypts it in the KDS's secret key, encrypts the conversation key 3 contained in the TGT (to be used later by the client to acquire service tickets) in conversation key 2, and returns this data to the client.
5. The client security runtime decrypts the reply from the authentication service by using conversation key 2, obtaining the conversation key 3 from the TGT, and it becomes part of the client's login context.

Note the following security safeguards inherent in the structure of this protocol:
- All network transmissions between the security client and the authentication service are encrypted by using strong random keys (not weak keys derived from passwords), placing even offline decryption attempts at the outer limits of practical possibility.
- The timestamp and conversation key 2 are encrypted by using the user's secret key, which is derived from the user's password (and subsequently reencrypted by using conversation key 1). This enables the authentication service to verify that the requesting client knows the user's password. (It does this by decrypting the

package via the registry service's copy of the user's secret key; if the decryption succeeds, the keys are the same, that is, they were derived from the same password.)

- The authentication service actively verifies whether the requesting client knows the user's password. Contrast this with the DCE Version 1.0 protocol, where the authentication service blindly issues TGTs without requiring any evidence that the requestor knows the user's password. It is therefore aware of, and can manage, persistent login failures for a given user, eliminating active password-guessing attacks.
- The authentication service's reply is encrypted by using conversation key 2, which was provided by the client. This verifies to the client that the authentication service itself is authentic since, if it were not, it would not have been able to obtain the machine session key and user's secret key it needed to decrypt conversation key 2.

These safeguards provide assurance to both server and client that the entity with which each is communicating is, in fact, what it claims to be.

Having acquired the user's TGT, the login program proceeds with the next step in the authentication procedure (described in "How the Client Obtains a PTGT for the User" on page 509).

The Timestamps Authentication Protocol: This section describes how the DCE Authentication Service uses the timestamps authentication protocol to provide a user with a TGT.

Since the timestamps protocol is largely identical to the DCE Version 1.0 protocol, which is fully explained in the next section, this section describes only the differences between the two.

The timestamps protocol proceeds exactly as the DCE Version 1.0 protocol described in "The DCE Version 1.0 Authentication Protocol" on page 507, with these additions:

- In Step 1, the client security runtime sends to the authentication service, in addition to the user's stringname, the current timestamp encrypted in the user's secret key.
- In Step 2, the authentication service, before preparing the user's TGT, verifies the user's authenticity (albeit not as strongly as in the third-party protocol) as follows:
  1. It decrypts the timestamp by using the copy of the user's key it obtained from the registry service.
  2. If the decryption succeeds, and the timestamp is within an allowable clock skew (5 minutes) of the current time, the user is authenticated, and the authentication service proceeds to prepare the TGT. If the decryption fails, or if the timestamp is not within the allowable clock skew, the authentication service rejects the login request.

With this protocol, the authentication service can verify the following:

- That the client login request is timely; that is, that the authentication service is communicating with the client now (within the allowable clock skew)
- That the requesting client knows the user's password

The authentication service is therefore aware of, and can manage, persistent login failures for a given user, eliminating passive password-guessing attacks.

From this point, the timestamps protocol continues as the DCE Version 1.0 protocol described in the next section, and then proceeds with the next step in the authentication procedure, described in “How the Client Obtains a PTGT for the User” on page 509.

Note: Encrypted timestamps (under the name authenticators) are passed in several places in the protocols, to guarantee fresh communications (within the allowable clock skew) and thereby guard against replay attacks. This has been shown explicitly in the preceding, but will be omitted in the remainder of this chapter.

The DCE Version 1.0 Authentication Protocol: This section explains how the DCE Authentication Service uses the DCE Version 1.0 protocol to authenticate a user. This protocol exists in DCE Version 1.1 solely to provide interoperability between DCE Version 1.1 servers and pre-DCE Version 1.1 clients; *only* pre-DCE Version 1.1 clients transmit DCE Version 1.0 login requests, and the authentication service returns DCE Version 1.0 responses *only* to pre-DCE Version 1.1 clients.

The DCE Version 1.0 protocol lacks the security features previously described for the third-party and timestamps protocols, hence this protocol is more vulnerable to attacks. You should keep this in mind when you are considering the inclusion of pre-DCE Version 1.1 clients in your DCE Version 1.1 cell.

The DCE Version 1.0 protocol proceeds as follows. Refer to Figure 71 on page 508 as you read these steps.

1. The user logs in, entering the correct user name. The login tool invokes **sec_login_setup_identity()**, which takes the user’s principal name as one of its arguments. This call causes the client security runtime to request a TGT and passes the user’s name (represented as a string, not a UUID) to the authentication service. The TGT will later be used by the client to acquire service tickets to other services; the first such usage will be to acquire a service ticket to the privilege service (see “How the Client Obtains a PTGT for the User” on page 509).

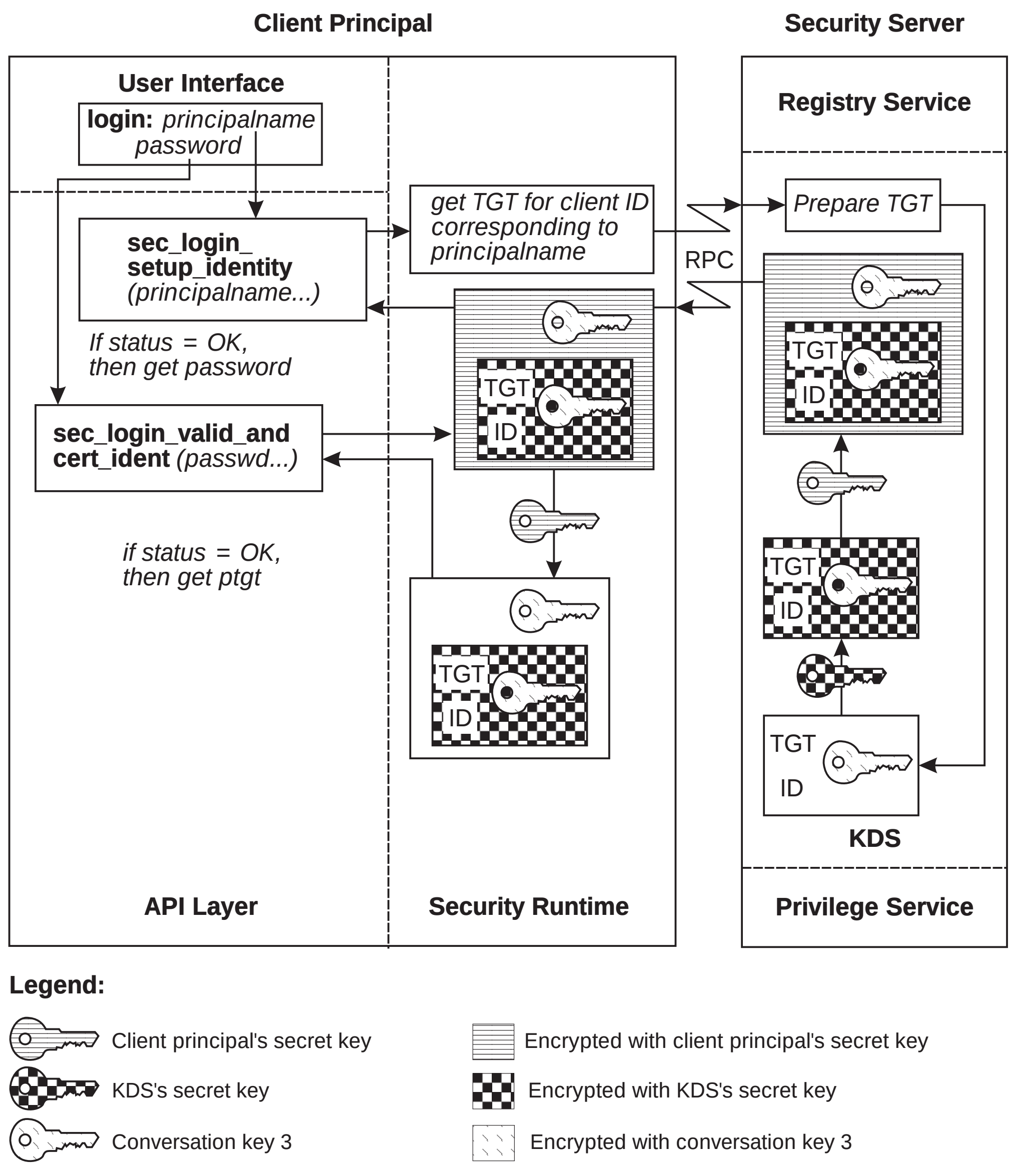

Client Principal
Security Server
User Interface
login: principalname password
Registry Service
get TGT for client ID corresponding to principalname
Prepare TGT
sec_login_ setup_identity (principalname...)
RPC
If status = OK, then get password
TGT
ID
sec_login_valid_and cert_ident (passwd...)
if status = OK, then get ptgt
KDS
API Layer
Security Runtime
Privilege Service
Legend:
Client principal's secret key
KDS's secret key
Conversation key 3
Encrypted with client principal's secret key
Encrypted with KDS's secret key
Encrypted with conversation key 3

Figure 71. Client Acquires TGT Using the DCE Version 1.0 Protocol

2. Upon receiving the request for a TGT, the authentication service obtains the user's secret key from the registry service database (where the secret keys of all principals in the cell are stored). Using its own secret key (that is, that of the KDS), the authentication service encrypts the user's identity, along with a conversation key 3 (this conversation key 3 is the same as conversation key 3 in the discussion of the third-party protocol, earlier in this chapter), in a TGT. The authentication service separately encrypts a copy of conversation key 3 with the user's secret key and returns this data to the client.
3. When this data arrives at the client, the login tool prompts the user for the password and invokes **sec_login_valid_and_cert_ident()**. This call passes the password to the client's security runtime library. The security runtime derives the user's secret key from the password (using a well-known algorithm), and uses it to decrypt conversation key 3. (If the user enters the wrong password, this decryption fails.) The client's security runtime cannot decrypt the TGT since it

does not know the KDS's secret key. The TGT is the client principal's certificate of identity—it is usable by the client precisely because the client knows the conversation key 3 carried in it.

Note: One of the functions of **sec_login_valid_and_cert_ident()** is to authenticate the authentication service itself to the host machine's login program, by demonstrating that the (purported) authentication service really knows the secret key of the host computer. (The mere fact that the purported authentication service knew the user's secret key is not convincing to the host's login program, because that purported authentication service could have been a bogus server working in league with a bogus user—the host doesn't trust any of these things.) The way in which this is accomplished is not illustrated here but is explained in "Chapter 30. The Login API" on page 581.

Having acquired the user's TGT, the login program proceeds with the next step in the authentication procedure, described in "How the Client Obtains a PTGT for the User".

How the Client Obtains a PTGT for the User

This section describes the acquisition, by the client's security runtime, of the user's PTGT. Acquisition of the user's PTGT is the second of the two parts of DCE user authentication.

From this point on, the client principal uses four different conversation keys to talk with other principals. Use of multiple short-lived keys makes an attacker's task far more difficult, since there are more encryption keys to discover and less encrypted material and time with which to crack them.

Refer to Figure 72 on page 511 as you read the following steps.

1.

   When the client's security runtime has succeeded in decrypting conversation key 3, it next wants to acquire a PTGT from the privilege service. Before a request for a PTGT can even be formulated, however, a service ticket to the privilege service must be acquired. The client's security runtime therefore begins by requesting such a service ticket from the TGS. The security runtime encrypts this request by using the conversation key 3 (which is also sealed in the client's TGT); it also sends along the client's TGT.
2. The TGS decrypts the TGT (which was encrypted in the KDS's secret key), learning conversation key 3, and verifies that the request was properly encrypted by using conversation key 3. This convinces the TGS that the identity of the requesting client is authentic; that is, no other principal could have sent a message so encrypted because no other principal knows conversation key 3. (The reader should review the preceding steps if necessary to be convinced that this is true.) Since the user has demonstrated to the TGS knowledge of the key, the TGS allows the user to talk to the privilege service, and so prepares a service ticket to that service. This ticket contains the identity of the user (and a conversation key 4), encrypted under the secret key of the privilege service (which the TGS retrieves from the registry service). The TGS separately encrypts conversation key 4 under conversation key 3, and returns this data to the client.

Note: Beginning with Figure 72 on page 511, the illustrations do not emphasize all the TGS's encryption and decryption activities (such emphasis would be redundant since the TGS knows all of the keys).

3. Upon receipt of this data, the client's security runtime uses conversation key 3 to decrypt conversation key 4. The client then formulates a request for a PTGT, encrypting it with conversation key 4, and sends this together with the service ticket it just received from the TGS, to the privilege service.
4. The privilege service decrypts the service ticket sent to it (using its secret key), thereby learning the identity of the client and the conversation key 4 it will use to decrypt the request and to encrypt its response. The privilege service is convinced of the authenticity of this request because the information was encrypted under its own secret key, and no principal other than the KDS (acting as the TGS) could have encrypted the information by using this secret key. Because the privilege service believes the authenticity of the client's identity, it prepares an extended privilege attribute certificate (EPAC) to issue to the client. (Actually, in the pure DCE Version 1.0 protocol this would be a PAC, not an EPAC, but since this is a high-level description intended for both releases we'll just talk about EPACs without fear of confusion. So what we're really describing here is an extended privilege TGT, or EPTGT, though we'll continue to call it a PTGT.)

   The EPAC describes the user's privilege attributes (identity information and group membership) and any extended attributes that are associated with the user—all represented as UUIDs (not strings). The EPAC (or EPAC chain, in case of a delegated operation) is sealed with an MD5 checksum. (Delegation is described in "Chapter 27. The Extended Privilege Attribute API" on page 537.) The privilege service constructs a PTGT, which is a ticket that contains the EPAC, the EPAC seal, another copy of the EPAC seal encrypted in the secret key of the privilege service (this is called a delegation token), and a conversation key 5 (which is actually generated by the KDS, though the illustration doesn't show this detail). All this information except for the EPAC itself is encrypted in the secret key of the KDS (thus, the delegation token is doubly encrypted). (The KDS and privilege service cooperate to prepare the PTGT, although the illustration only shows the privilege service preparing it.) The EPAC seal inside the PTGT binds the EPAC to the PTGT, guaranteeing its integrity even though it isn't encrypted. The conversation key 5 is encrypted in conversation key 4, and all this data is returned to the client.

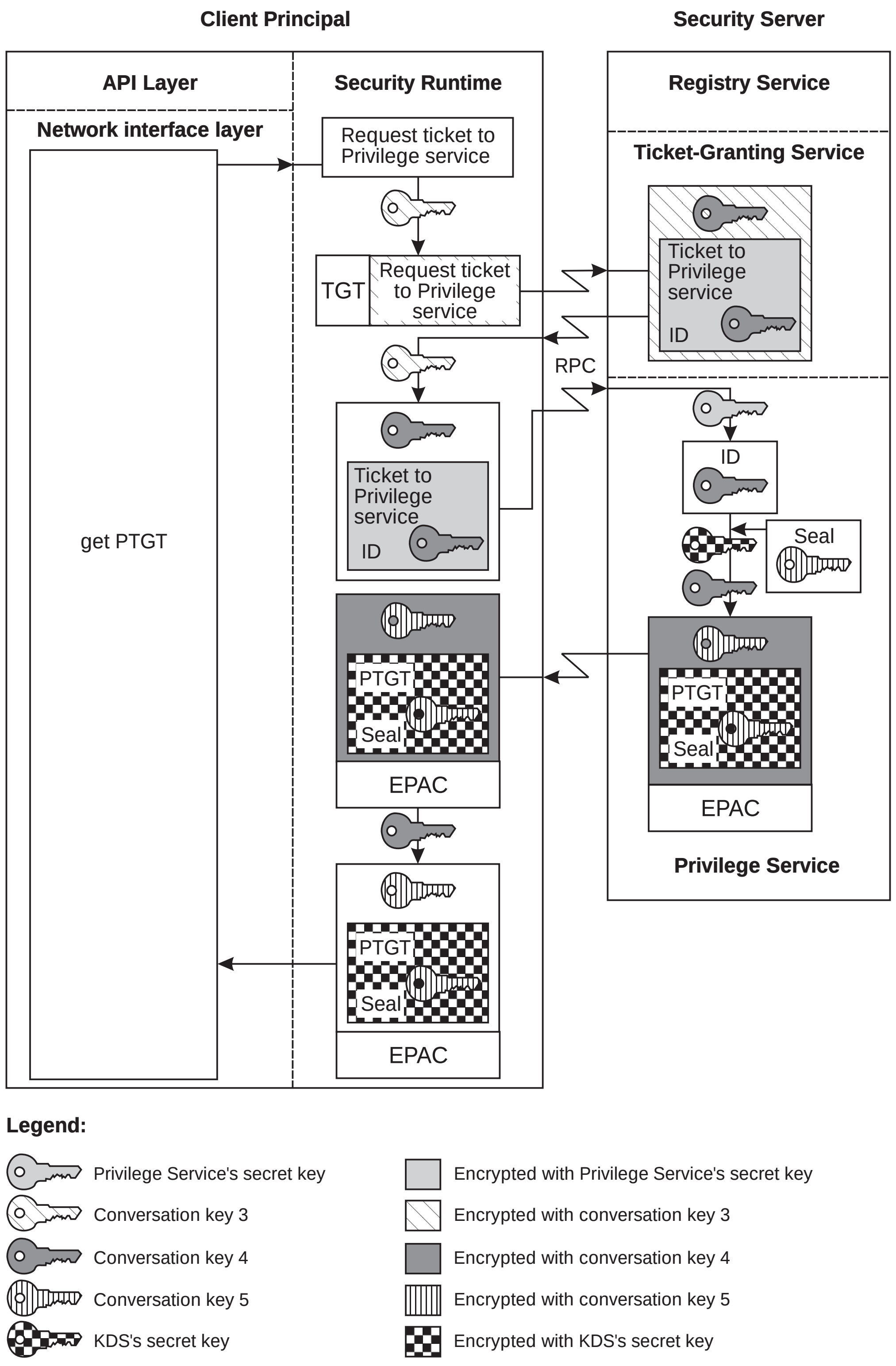

Client Principal
Security Server
API Layer
Security Runtime
Registry Service
Network interface layer
Request ticket to Privilege service
Ticket-Granting Service
TGT
Request ticket to Privilege service
Ticket to Privilege service
ID
RPC
ID
get PTGT
Ticket to Privilege service
ID
Seal
PTGT
Seal
EPAC
PTGT
Seal
EPAC
Privilege Service
PTGT
Seal
EPAC
Legend:
Privilege Service's secret key
Conversation key 3
Conversation key 4
Conversation key 5
KDS's secret key
Encrypted with Privilege Service's secret key
Encrypted with conversation key 3
Encrypted with conversation key 4
Encrypted with conversation key 5
Encrypted with KDS's secret key

Figure 72. Client Acquires PTGT

5. The client's security runtime uses conversation key 4 to decrypt conversation key 5. It cannot decrypt the PTGT itself, since the PTGT is encrypted under the secret key of the KDS.

The Login Context

At this point, the security service has authenticated the user's identity (that is, has verified that the user knows its password), and the user has acquired (trusted) information about its privilege attributes from the privilege service. The client now calls **sec_login_set_context()** to set the login context (a handle to this user's network identity and privilege attributes that have been established). Henceforth, processes invoked by this user inherit the user's login context, and among these processes is the client side of distributed applications — those are the subject of the rest of the walkthrough.

Identities in a Delegation Chain

When a user who has initiated delegation (with **sec_login_become_initiator()**) makes an authenticated RPC to the next member in a delegation chain (the first intermediary), the initiator passes its PTGT (including EPAC, seal and delegation token) to the TGS, and receives an extended privilege service ticket (again containing EPAC, seal and delegation token) to the intermediary. This is passed to the intermediary. The intermediary then invokes either routine **sec_login_become_delegate()** or **sec_login_become_impersonator()**, passing to the privilege service the authorization information it received from the initiator (EPAC and delegation token), together with the intermediary's own PTGT (including the intermediary's EPAC, seal and delegation token).

The privilege service uses the two delegation tokens, which are seals over the initiator's and intermediary's EPAC encrypted in the privilege service's own secret key, to verify the authenticity of the EPACs. If these are valid, the privilege service creates an EPAC chain, consisting of the initiator's and intermediary's EPACs, and generates a new seal and delegation token for this EPAC chain, and returns to the intermediary a new PTGT containing this information. Thus, the intermediary's authorization information now includes both EPACs in the delegation chain and a PTGT that contains the EPAC chain's seal and delegation token. The subsequent additions of identities to the delegation chain are handled in the same manner, resulting in PTGTs with each intermediary's identity being added to the EPAC chain. Any such PTGT can be used to continue the delegation chain or to acquire a service ticket to the ultimate target server.

Authenticating an Application

Applications that are run between client and server must also be authenticated. For specific information about using the authenticated RPC routines see "Chapter 13. Basic RPC Routine Usage" on page 183 and "Chapter 14. RPC and Other DCE Components" on page 195. For information about the GSSAPI, see "Chapter 23. Overview of Security" on page 483 and "Chapter 26. GSSAPI Credentials" on page 533.

Authentication Using Authenticated RPC

This section explains how DCE security authenticates an application, to which the application developer has added authenticated RPCs.

Note: The authenticated RPC facility may also be referred to as the *protected* RPC facility, as it involves services beyond authentication. Authenticated RPC may also be referred to as the protected RPC facility, Refer to Figure 73 as you read the following steps.

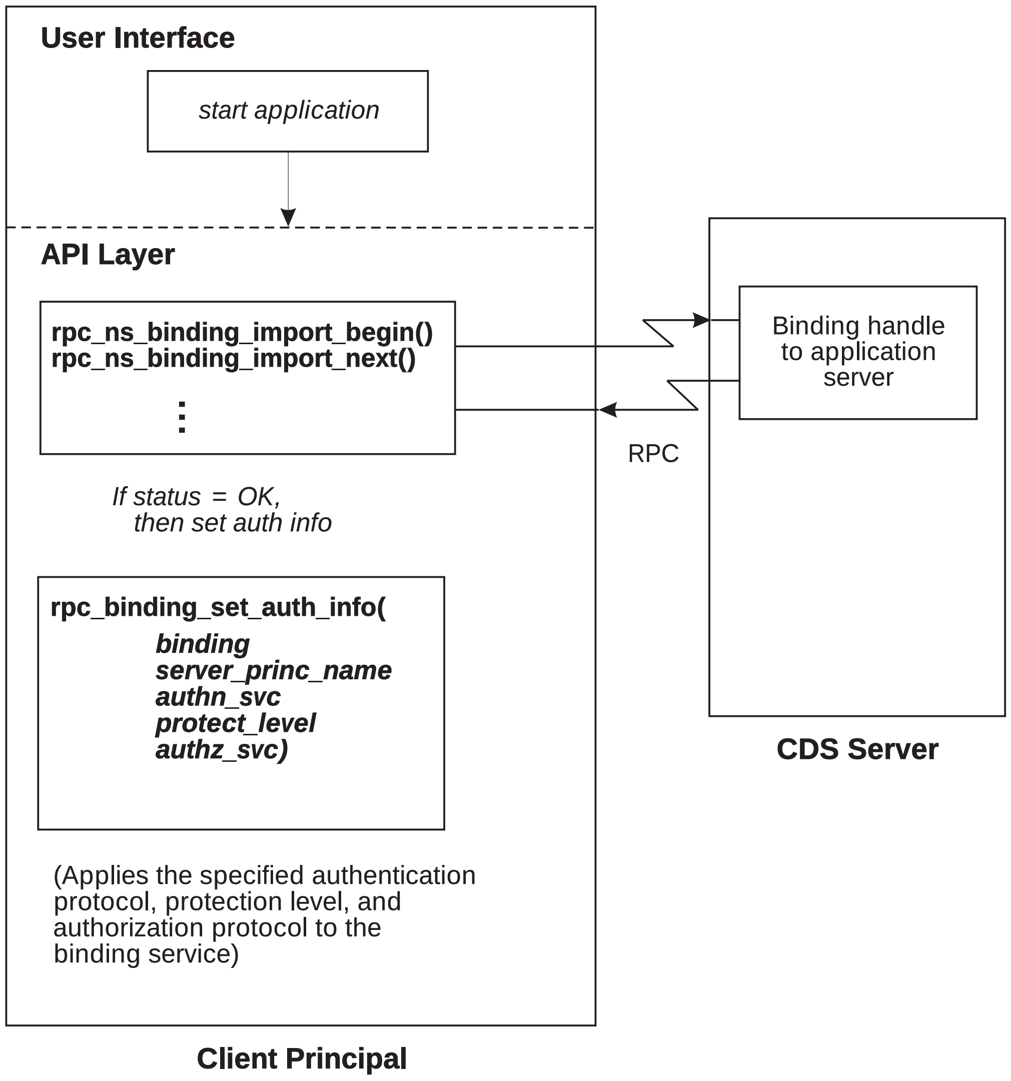

User Interface
start application
API Layer
rpc_ns_binding_import_begin()
rpc_ns_binding_import_next()
⋮
If status = OK, then set auth info
rpc_binding_set_auth_info(
binding
server_princ_name
authn_svc
protect_level
authz_svc)
(Applies the specified authentication protocol, protection level, and authorization protocol to the binding service)
Client Principal
Binding handle to application server
RPC
CDS Server

Figure 73. Client Sets Authentication and Authorization Information

1. Having been authenticated and having acquired a PTGT, the user invokes an application. The client side of the application makes calls to routines **rpc_binding_import_begin()**, **rpc_binding_import_next()**, and the like. These calls specify the remote interfaces required by the client for the application.
2. The CDS returns the client binding handles to the specified interfaces. (For simplicity in this example, we consider the simple binding model in which the client consults the CDS for the server's RPC binding name.)
3. The client *annotates the binding handle*—that is, it sets security information for the binding handle by calling **rpc_binding_set_auth_info()**. Among other parameters, this routine sets the authentication protocol, the protection level, and authorization protocol for the binding handle corresponding to the remote interface. It also sets the server's principal name, which the client must know securely (it may be the same or different than the server's RPC binding name). In this example, assume that the authentication protocol (*authn_svc* parameter) is DCE shared-secret authentication, the protection level (*protect_level*) is packet privacy (all RPC argument values are encrypted), and the authorization

protocol (*authz_svc*) is DCE authorization. (DCE authorization means that an EPAC chain, containing UUIDs representing the client's or delegation chain's privilege attributes, will be sent to the server, which will compare this information with the ACLs protecting the objects of interest in order to determine whether the principal is to be granted or denied access.)

Refer to Figure 74 on page 515 as you read the following steps.

4. The client requests some operation (using the annotated binding handle) to be performed by the server. The client RPC runtime requests from the TGS a service ticket to the server (identified by the server principal name with which the binding handle has been annotated). To acquire the ticket, the client security runtime formulates a request to the TGS. The request includes the server's principal name, which the client security runtime encrypts under conversation key 3. Also sent along with the request is the principal's PTGT, including EPAC and seal.

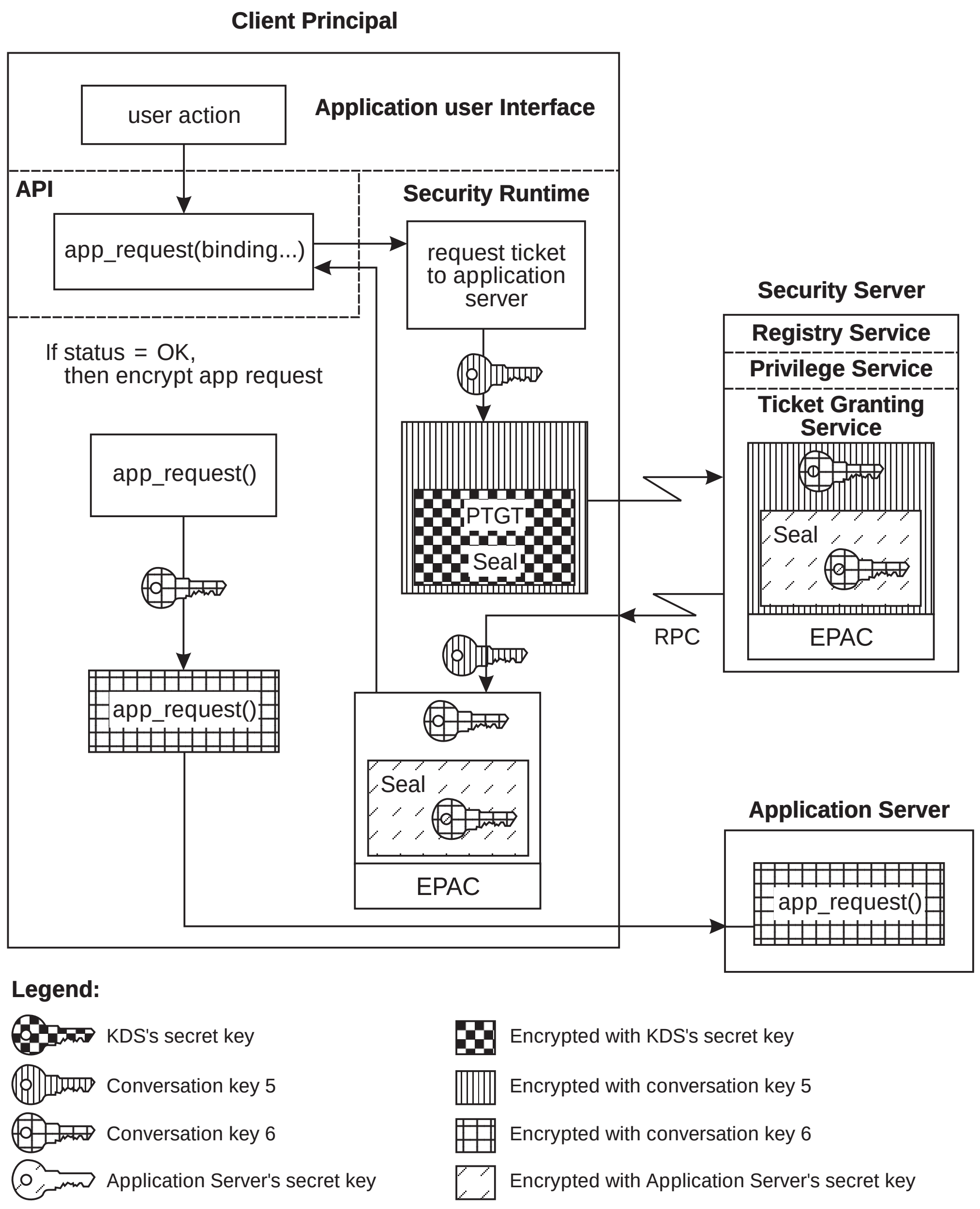


Figure 74. Client Principal Makes Application Request

5. The TGS decrypts the PTGT (which was encrypted in the KDS's secret key), thereby recovering conversation key 5, and uses conversation key 5 to decrypt the rest of the TGS request message. The TGS then constructs a service ticket, including the EPAC chain information and conversation key 6. By default, the key that is used to encrypt the service ticket is the application server's secret key.

   For server principals that must use the user-to-user authentication protocol, the service ticket granted must be encrypted using the session key obtained from the server's current TGT, which the client must pass in with the ticket request. If the client had used the server-key-based request, and the server requires user-to-user protocol, the TGS will respond with an error instructing the client-side runtime to ask the server for its current TGT and to reissue the request with this TGT.

The service ticket is returned to the client, together with conversation key 6 encrypted under conversation key 5.

6. The client’s security runtime uses conversation key 5 to decrypt conversation key 6, and then uses conversation key 6 to encrypt the application-level RPC request to the server. The client’s RPC runtime sends the encrypted application request to the application server, together with the service ticket.

   Refer to Figure 75 on page 517 as you read the following steps.

7. The application server’s security runtime receives the client’s request and decrypts the service ticket by using its secret key, or the TGT session key if user-to-user based authentication is used. In this way, the server’s security runtime learns conversation key 6 and uses it to decrypt the RPC request. If the server determines from the client’s authorization information (EPAC chain) that the request is granted, it performs the requested operation and prepares a response. The server’s runtime encrypts the response by using conversation key 6 and sends it back to the client.
8. The client runtime receives and decrypts the response, and returns data to the application (by returning from the RPC).

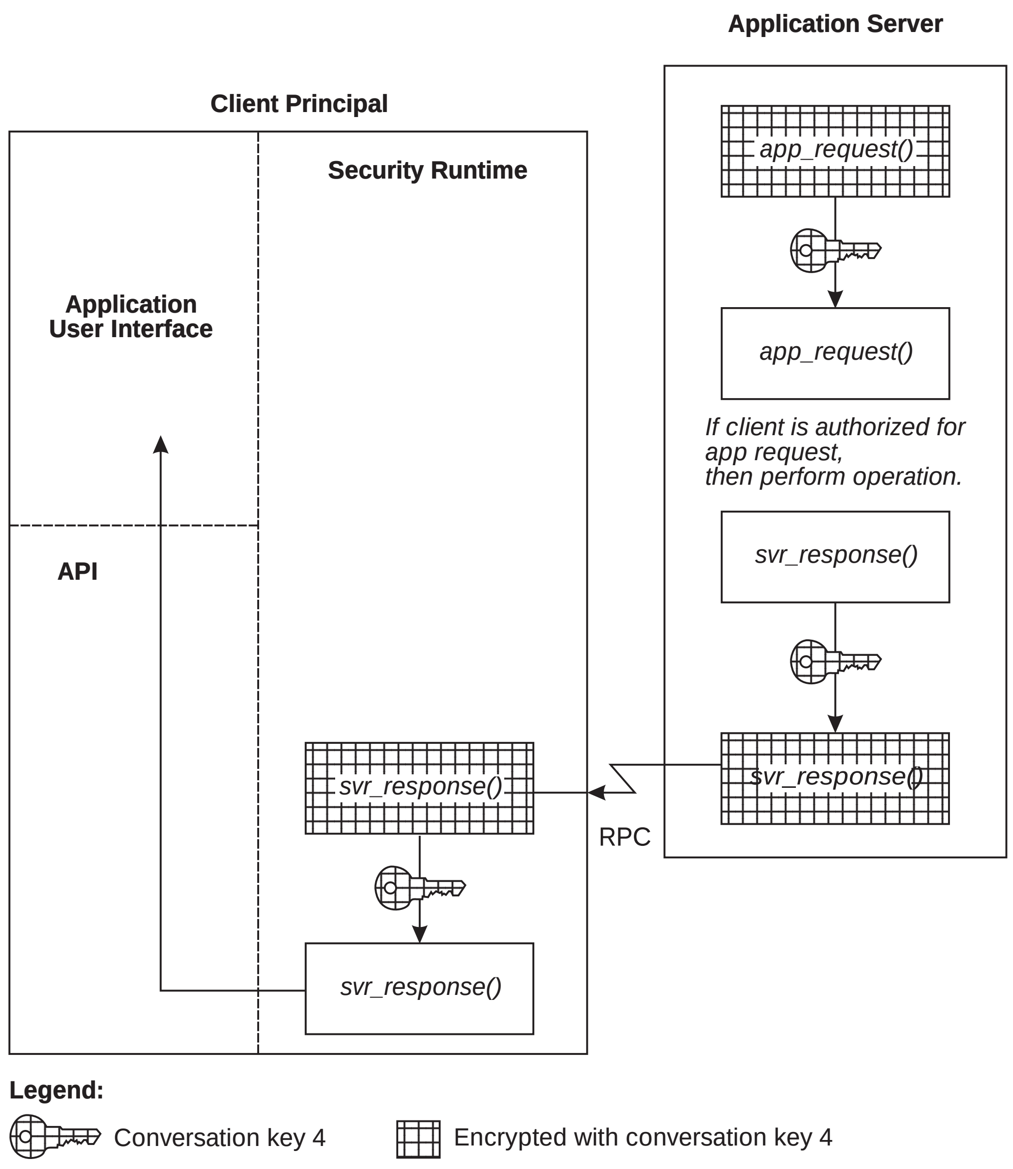


Figure 75. Application Server Responds to Client's Request

The preceding walkthroughs have focused on the security aspect of authenticated RPC in DCE, not on its communications aspect. The technical details of integrating security with RPC lie beyond the scope of this chapter. However, the following remarks apply:

1. In the CO (virtual circuit) RPC protocol, the client's security credentials (ticket with conversation key 6 and EPAC) are *pushed* from client to server at connection establishment time, that is, at the time of the first remote procedure call from client to server (and the remote call is of course protected with conversation key 6). In the CL (datagram) RPC protocol, on the other hand, while the first remote call from client to server is protected as previously described (with conversation key 6), the credentials themselves are not sent with the remote call. Instead, the server itself pulls the credentials, by performing a callback, that is, a reverse (system-level) RPC back to the client, requesting the credentials. Once it receives these credentials, the server

proceeds as if the credentials had been transferred with the original application-level RPC (from client to server) itself, as in the preceding walkthroughs.

2. Once the application client and server have established conversation key 6, they cache it and continue to use it for subsequent RPCs, until it *expires*. All tickets and their conversation keys are accompanied by an expiration time, beyond which a new conversation key must be established (via a new service ticket, or perhaps even a new TGT if that expires, as described in the preceding walkthroughs). Thus, the security overhead of these subsequent RPCs is minimal, namely, it is reduced merely to the overhead of encryption/decryption processing itself, without the protocol message-passing.

Authentication Using GSSAPI

This section describes the process by which applications that perform their network communications via a mechanism other than DCE RPC can use GSSAPI and DCE security to authenticate and otherwise protect their communications. (These alternative communications mechanisms are called peer-to-peer, to distinguish them from RPC.)

In peer-to-peer communications, the application component that establishes the secure connection is called the context initiator or simply initiator. The context initiator is analogous to a DCE RPC client. The application component that accepts the secure connection is called the context acceptor or simply acceptor. The context acceptor is analogous to a DCE RPC server.

The peer application components establish a secure connection in the following way. (The reader will notice that the underlying security aspects are identical to those of the preceding RPC case, the only differences being in the explicit routine-invocation and communications aspects.)

1. The context initiator uses the **gss_init_sec_context()** routine to request from the DCE security server a service ticket (as previously described) that will allow the initiator to talk to the context acceptor.

   The initiator's security runtime creates an envelope that contains:

   - The initiator's PTGT

     Note: It is assumed that the initiator's security runtime already possesses a PTGT; that is, GSSAPI itself does not handle login.

   - The acceptor's principal name, protected under conversation key 5

   The initiator's security runtime sends the envelope to the TGS. (As in "Authentication Using Authenticated RPC" on page 512, step 4, this communication happens via RPC, but this use of RPC is hidden from the application because it's an implicit RPC being made by the security runtime, not an explicit RPC by the application initiator itself.) The TGS issues a service ticket to the initiator, encrypted in the acceptor's secret key, exactly as described in "Authentication Using Authenticated RPC" on page 512, step 5.

2. The initiator's security runtime recovers conversation key 6 as described in "Authentication Using Authenticated RPC" on page 512, step 6, and then hands to the GSSAPI the service ticket (including EPAC chain) and conversation key 4.

3. GSSAPI holds onto conversation key 6 and creates a GSSAPI *token* containing the service ticket.

This GSSAPI token is then returned to the initiator, which forwards it to the acceptor (via the application's chosen communications mechanism). (Compare this with "Authentication Using Authenticated RPC" on page 512, step 6.)

4. The acceptor calls the **gss_accept_sec_context()** routine, which passes the token to the acceptor's security runtime.
5. The acceptor's security runtime processes the token, in particular recovering conversation key 6, exactly as described in "Authentication Using Authenticated RPC" on page 512, step 7.
6. The acceptor's GSSAPI holds onto conversation key 6 and the EPAC chain, and creates a GSSAPI token containing the success message. It passes the token to the acceptor. (Again, refer to "Chapter 24. Authentication" on page 493, step 7.)
7. The acceptor forwards the GSSAPI token to the initiator.
8. The initiator passes the token to its GSSAPI, which sends it to the security runtime by calling the **gss_init_sec_context()** routine again.
9. The initiator's security runtime tries to decrypt the message. If this succeeds, it returns a success status to the GSSAPI that the acceptor's identity is authenticated. If not, it returns a failure status to the GSSAPI. (Compare this to "Chapter 24. Authentication" on page 493, step 8.)

The context acceptor and context initiator can then use conversation key 6 in future communications by calling the **gss_sign()** and **gss_seal()** routines. (Compare this scenario with the RPC remarks following Section "Authentication Using Authenticated RPC" on page 512, step 8.) The context acceptor can get the initiator's EPAC chain in the form of an **rpc_authz_cred_handle_t** object so it can perform a DCE ACL check by calling the **gssdce_extract_creds_from_sec_context()** routine. If the context initiator wants to talk to a different context acceptor, it must acquire a ticket to that context acceptor.

Intercell Authentication

While the intercell authentication model is an extension of intracell authentication, certain concepts are particular to intercell authentication. The following subsections discuss those concepts.

KDS Surrogates

A principal trusts the DCE Security Service (registry service/KDS/privilege service) to authenticate other principals in its cell because it trusts the cryptographic algorithms and protocols, and the security of the code and data of the security service itself (which is trusted because it is part of the DCE network trusted computing base). The DCE Security Service can authenticate all principals in its cell because it shares a secret key with each of them. A client principal that wants to talk to a *foreign* server principal (that is, a principal in another cell) must acquire a ticket targeted to that server. As always, such a ticket must be encrypted in the secret key of the foreign server, else the server will not trust the ticket. The client cannot get such a ticket from its own local security service, because only the foreign security service, not local security service, knows the secret key of the foreign server. Therefore, some means must be devised by which the two instances of the security service can securely convey information about their respective principals to one another (without actually divulging secret keys of principals to foreign security services, which would be a security risk).

Besides the fact that it is trusted a priori, a cell’s KDS is an exceptional principal in this other respect: other kinds of principals share their secret keys with the local security service, whereas the KDS’s key is private to the KDS; that is, it is known to no other principal. Thus, one problem that intercell authentication must overcome is the means by which the KDS in one cell may trust that in another cell without either of them having to share their private keys (which would again introduce an unacceptable security risk).

Note: With respect to cryptographic keys, the term secret refers to keys that are (securely) shared between a bounded set of two (or more) principals, while private refers to keys that are known to only a single principal, and public to keys that are known to an unbounded set of principals (potentially to all principals). The cryptographic algorithms and protocols that are currently supported by DCE all depend on secret key technology (typified by DES), even though a small number of private keys (those of KDSs) are used.

The solution to this problem is a small extension of the shared-secret authentication model previously discussed in this chapter. Namely, a new principal is invented specifically for cross-cell authentication, and two entries for this principal are made, one each in the registry service databases of the two mutually authenticating cells. The two entries have the *same* secret key. These two special registry service database entries are known as mutual authentication surrogates, and the two cells that maintain mutual authentication surrogates are called trust peers. It is through their surrogates that the two instances of the KDS can convey information about their respective principals to one another (though the two KDSs never communicate directly with one another, nor do the surrogates), thus enabling a client principal from one cell to acquire a ticket to a server principal in another cell.

An authentication surrogate is a true principal in the sense that it is represented by an entry in a registry service database, but it is not an autonomous participant in authenticated communications in the same sense that, for example, a client or a server is. Rather, it is more like an alias that is assumed by a cell’s KDS when it communicates with foreign clients. The establishment, via surrogates, of a trust peer relationship between two cells is an explicit expression of mutual trust in the two KDSs on the part of the cell administrators who establish the relationship. Administrators use the **rgy_edit** tool to create surrogates and establish the trust relationship. Administrators who do not trust one another’s cells must not establish such a relationship.

Intercell Authentication by Trust Peers

This section explains how a client principal in one cell is authenticated by the KDS in a peer cell, so that the client principal may communicate with a server principal that is a member of the foreign cell. The style of description is the same as in the walkthroughs earlier in this chapter, though no figures are used here.

1. A client principal, having already been authenticated in the normal way by the KDS and privilege service in its home cell and acquired its PTGT, requests its local TGS for a service ticket targeted to a server in a foreign cell. The client specifies the server principal by its fully qualified principal name, which includes the name of the foreign cell.
2. The client’s security runtime makes a request to the client’s local TGS for a service ticket to the foreign server. The TGS recognizes by the server’s principal name that it is foreign, so this TGS cannot directly issue the desired service ticket. Instead, it issues a so-called cross-cell TGT (XTGT), which is targeted to the surrogate shared between the two cells (that is, it is encrypted in the

surrogate's secret key). The EPAC data in the client's PTGT is copied into the XTGT, and the local TGS returns the XTGT to the client. (For simplicity, we deal here only with simple case of EPAC data, not a delegation EPAC chain.)

3. The client receives the XTGT, recognizes that it is not targeted to the application server it had requested, and proceeds to send a request to the foreign TGS for a service ticket to the foreign privilege service, this time presenting the XTGT (instead of its original TGT) as proof of authentication. Upon receiving this request, the foreign TGS decrypts it by using the surrogate's secret key, and returns to the client a service ticket to the foreign privilege service. (Note how knowledge of the surrogate's shared key makes it possible for the two TGSs to cooperate in this way.)
4. The client's security runtime sends this service ticket to the foreign privilege service, to obtain a cross-cell privilege TGT (XPTGT). This XPTGT contains the client's original EPAC, and is encrypted with the secret key of the foreign privilege service.
5. After the client principal receives the XPTGT, it sends it to the foreign TGS, requesting a service ticket to the foreign server principal it was originally interested in. From this point on, the protocol goes exactly as it would in the case of a client principal in the server's cell requesting a service ticket to that server (as previously described). Similarly, the client principal may reuse the XPTGT to acquire service tickets to any other servers in the foreign cell.

Chapter 25. Authorization

This chapter explains concepts related to authorization. The authenticated RPC facility enables you to select the authorization protocol that your application uses. Among the authorization protocols supported by the DCE Security Service for use by authenticated RPC is DCE authorization (the default), and name-based authorization.

This chapter first discusses DCE authorization, and more particularly, DCE access control lists (ACLs). At the end of this chapter, we also briefly discuss the name-based authorization protocol.

DCE Authorization

The DCE authorization protocol is based in part on the UNIX file-protection model, but is extended with ACLs. An ACL is a list of access control entries that protects an object. Each entry in the ACL specifies a set of permissions. Usually, most of the entries in the ACL specify a privilege attribute (such as membership in a group) and the set of permissions that may be granted to the principal(s) that possesses that privilege attribute. Some other entries specify a set of permissions that may mask the permission set in a privilege attribute entry.

Every ACL is managed by an ACL manager type. An ACL manager type determines a principal's authorization to perform an operation on an object by reading the object's ACL to find the appropriate entry (or entries) that matches some privilege attribute possessed by the principal. If the type of access requested by the principal is one of the permissions listed in the matching entry, and assuming no applicable mask entry denies that permission, then the ACL manager type allows the principal to perform the requested operation. If the requested permission is not listed in the matching ACL entry, or is denied by a mask, permission to perform the operation is denied. Permission to perform the operation is also denied if the ACL contains no matching privilege attribute entry.

Unlike UNIX file permissions, DCE ACLs are not limited to the protection of file system objects such as is, files, directories, and devices. ACLs may also control access to nonfile-system objects, such as the individual entries in a database.

Note: The implementation of DCE ACLs is aligned with POSIX P1003.6 Draft 12.

In the discussions in this chapter, we use the general term name to refer to a principal, group, or cell identifier; but readers should always bear in mind that these names have two representations: as UUIDs in ACL program interfaces and as print strings in user interfaces.

Object Types and ACL Types

The ACL facility distinguishes between two types of objects: container objects and simple objects. Container objects contain other objects, which may be simple and/or other container objects. Simple objects do not contain other objects. Examples of container objects include file-system directories and databases; examples of simple objects include files and database entries.

To protect both object types, and to enable newly created objects to inherit default ACLs from their parent container objects, the ACL facility supports two basic kinds of ACLs:

- An Object ACL is associated with either a container or a simple object, and controls access to it.
- A Creation ACL is associated with a container object only. Its function is not to control access to the container but to supply default values for the ACLs of objects created in the container. There are two types of Creation ACLs:
  - An Initial Object Creation ACL supplies default values for a simple object's Object ACL and for a container object's Initial Object Creation ACL.
  - An Initial Container Creation ACL supplies default values for both a container object's Object ACL and its Initial Container Creation ACL.

Figure 76 illustrates how ACL defaults are derived from Creation ACLs.

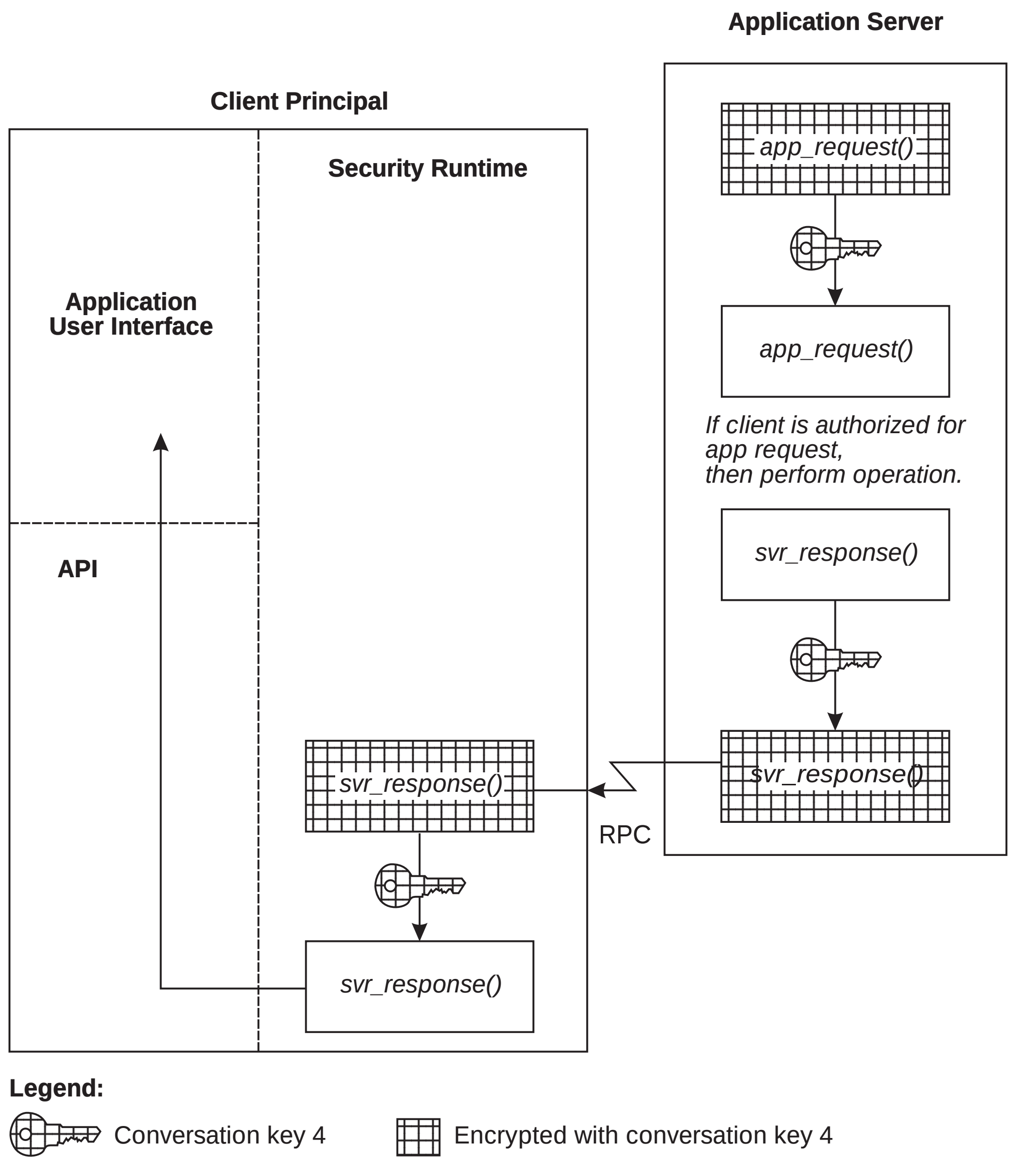


Figure 76. Derivation of ACL Defaults

Aside from the distinctions previously described, there are no differences between Object ACLs and Creation ACLs; therefore, the information about ACLs in the rest of this chapter does not differentiate between them.

ACL Manager Types

A separate ACL manager type manages the ACLs for each class of objects for which permissions are uniquely defined. The manager type defines the permissions for those objects whose ACLs it manages, which are the number of permissions, the meanings of the permissions, and the tokens that represent the permissions in user interfaces to ACL manipulation tools.

For example, for the purpose of access control, five classes of objects are defined in the registry database, and five ACL manager types manage the ACLs for the registry database objects (the five registry manager types run in a single security server process). Other DCE components implement their own manager types, and applications implement manager types for the objects that the applications protect.

Refer to the *OSF DCE Administration Guide* and the *OSF DCE Administration Commands Reference* for information about standard DCE ACL manager types and the permissions they implement. Refer to “Part 1. DCE Facilities” on page 1 and “Chapter 32. The Access Control List APIs” on page 591 of this guide for information about implementing ACL manager types for distributed applications.

Access Control Lists

An ACL consists of the following:

- An ACL manager type identifier, which identifies the manager type of the ACL.
- A default cell identifier, which specifies the cell of which a principal or group identified as local is assumed to be a member. A DCE global pathname is necessary to specify a principal or a group from a nondefault cell; this consists of a pair of UUIDs representing the principal or group, and the cell of which it is a member. It is necessary to use the ID Map API to convert the global print string names of foreign principals and groups to the UUID representations that DCE ACL managers use. (Refer to “Chapter 33. The ID Map API” on page 601 for more information on this subject.)
- At least one ACL entry.

The rest of this chapter discusses ACLs primarily from a user-interface point of view, since this perspective provides an orientation to the discussion of the ACL API in this part.

ACL Entries

DCE authorization defines two basic kinds of ACL entries:

- Those that associate a specified privilege attribute with a permission set; these are privilege attribute entries.
- Those that specify a permission set that masks a permission set specified in a privilege attribute entry; these are mask entries.

The following subsections describe the two kinds of ACL entries in detail.

Privilege Attribute Entry Types

The privilege attributes of a principal are based on identity and include the principal's name, its group membership(s), and native cell. Note that not all ACL manager types implement all privilege attribute entry types. For example, the ACL manager type of a database object probably would not support the **user_obj** and **group_obj** entry types.

Note: The term local cell means the cell specified in the ACL header; this is not necessarily the cell in which the protected object resides.

The descriptions of the ACL entry types that specify privilege attributes are as follows:

- **user_obj**

  The **user_obj** entry establishes the permissions for the object's "user" (in the established UNIX sense). An ACL may contain only one entry of this type. The identity of the principal to which this ACL entry refers is assumed to be local and is specified somewhere other than in this entry. In the case of a file, for example, the identity is attached to the file's inode.
- **user**

  The **user** entry establishes the permissions for the local principal named in this entry. An ACL may contain a number of entries of this type, but each entry must be unique with respect to the principal it specifies.
- **foreign_user**

  The **foreign_user** entry establishes the permissions for the foreign principal named in this entry. An ACL may contain a number of entries of this type, but each entry must be unique with respect to the foreign principal it specifies. This entry type is exactly like the **user** entry type, except that this entry explicitly names a cell. (For the entry type **user**, the principal inherits the cell specified by the default cell identifier in the ACL header.)
- **group_obj**

  The **group_obj** entry establishes the permissions for the object's "group" (in the established UNIX sense). An ACL may contain only one entry of this type. As is the case with the **user_obj** entry, the identity of the group is assumed to be local and is specified elsewhere than in the **group_obj** entry itself.
- **group**

  The **group** entry establishes the permissions for the local group named in this entry. An ACL may contain a number of entries of this type, but each entry must be unique with respect to the group it specifies.
- **foreign_group**

  The **foreign_group** entry establishes the permissions for the foreign group named in this entry. An ACL may contain a number of entries of this type, but each entry must be unique with respect to the foreign group it specifies. This entry type is exactly like the **group** entry type, except that this entry explicitly names a cell (for the entry type **group**, the principals inherit the default cell identifier).
- **other_obj**

  The **other_obj** entry establishes the permissions for local principals whose identities do not correspond to any entry type that explicitly names a principal or group; an ACL may contain only one entry of this type.
- **foreign_other**

The **foreign_other** entry establishes the permissions for all principals that are members of a specified foreign cell and whose identities do not correspond to any **foreign_user** or **foreign_group** entry. An ACL may contain a number of entries of this type, but each entry must specify a different foreign cell.

- **any_other**

  The **any_other** entry establishes the permissions for principals whose privilege attributes do not match those specified in any other entry type. An ACL may contain only one entry of this type.

The following additional ACL entry types are supplied for delegated identities:

- **user_obj_delegate**
- **user_delegate**
- **foreign_user_delegate**
- **group_obj_delegate**
- **group_delegate**
- **foreign_group_delegate**
- **foreign_other_delegate**
- **other_obj_delegate**
- **foreign_other_delegate**
- **any_other_delegate**

These ACL entry types are described in detail in “Chapter 27. The Extended Privilege Attribute API” on page 537, along with the extensions to the ACL checking algorithm for delegation.

ACL entries for privilege attributes consist of three fields in the following form:

```
entry_type[: key]: permissions
```

Following are descriptions of the fields:

- The ACL *entry_type* specifies an ACL entry type as described in the previous list.
- The *key* field specifies the privilege attribute to which the permissions listed in the entry apply. The key field for the ACL entry types **user**, **group**, **foreign_user**, **foreign_group**, and **foreign_other** explicitly names a principal, group, or cell. For the entry types **foreign_user**, **foreign_group**, and **foreign_other**, the key field must contain a global DCE pathname of the forms ***/.../*** *cellname/principalname*, ***/.../*** *cellname/groupname*, or ***/.../*** *cellname*, respectively. The entry types **user_obj**, **group_obj**, **other_obj**, and **any_other** do not use the key field.
- The *permissions* field lists the permissions that may be granted to the principal possessing the privilege attribute specified in the entry, unless a mask (or masks) further restricts the permissions that may be granted to the principal. As noted previously, the number and meaning of the permissions that may protect an object are defined by the object’s ACL manager type. Therefore, the permissions that an ACL entry may specify must be the set, or a subset, of the permissions implemented by the manager type of the ACL in which the entry appears.

  A principal is denied access when a **user**or **foreign_user** entry that names the principal contains an empty permission set.

Mask Entry Types

Following are descriptions of the ACL entry types that specify masks:

- **mask_obj**

  The **mask_obj** entry establishes the permission set that masks all privilege attribute entry types except the **user_obj** and **other_obj** types.
- **unauthenticated**

  The **unauthenticated** entry establishes the permission set that masks the permission set in a privilege attribute entry that corresponds to a principal whose privilege attributes have not been certified by an authority such as the privilege service.

The two masks are similar in that the permission set specified in the mask entry is intersected (logically ANDed) with the permission set in a privilege attribute entry. This masking operation yields the effective permission set (the permissions that may be granted to the principal) for the principal possessing the privilege attribute. For example, if a privilege attribute entry specifies the permissions **ab**, and a mask entry that specifies the permissions **bc** masks that privilege attribute entry, the effective permission set is **b**. Similarly, a mask entry that specifies the empty permission set means that none of the permissions in any privilege attribute entry that mask entry masks is granted to the principal possessing the privilege attribute.

The two masks are dissimilar in one notable respect. Adding an **unauthenticated** mask entry with an empty permission set to an ACL is equivalent to omitting the **unauthenticated** mask entry from the ACL; in both cases, the set of effective permissions for principals possessing unauthenticated privilege attributes is empty. However, adding a **mask_obj** entry with an empty permission set to an ACL is different from having no **mask_obj** entry in the ACL. In the first case, the effective permission set is empty; in the second case, the effective permission set is identical to the permission set in the privilege attribute entry.

ACL entries for masks consist of two fields in the following form:

```
entry_type:permissions
```

Following are descriptions of the fields:

- The *entry_type* field specifies one of the two masks entry types: **mask_obj** or **unauthenticated.**
- The *permissions* field specifies the permission set that masks the permission set in any privilege attribute entry masked by the mask entry.

The Extended ACL Entry Type

The ACL entry type **extended** is a special entry type for ensuring the compatibility of ACL data created by different software revisions. It enables old application clients to copy ACLs from one newer revision object store to another without losing data. It also enables obsolete clients to manipulate ACL data that they understand without corrupting the extended entries that they do not understand.

Access Checking

Standard DCE ACL manager types use a common access-check algorithm to determine the permissions they grant to a principal. Access checking is executed in up to six stages, in the following order:

1. The **user_obj** entry check
2. The check for a matching **user** or **foreign_user** entry
3. The **group_obj** entry check and the check for matching **group** or **foreign_group** entries
4. The **other_obj**entry check
5. The check for a matching **foreign_other** entry
6. The **any_other** check

If during any stage of access checking an ACL manager type finds a privilege attribute entry that matches a privilege attribute possessed by a principal, then the manager type does not execute any subsequent stages, even though the principal may possess other privilege attributes for which there are other matching entries. See the *Security Volume* of the *Application Environment Specification/Distributed Computing* for descriptions of the algorithms used at each stage of access checking.

Examples of ACL Checking

The following subsections provide some examples that illustrate ACLs and the access-check algorithms. The examples use the arbitrary convention of ordering entries as follows: masks, principals, groups, and ″other'' entries. However, the access check algorithm disregards the order in which entries appear in an ACL. Also note that the permissions in these examples do not refer to any particular permissions implemented by any ACL manager type.

Example 1

Following is an ACL that protects an object to which three principals, **janea**, **/.../cella/fritzb**, and **mariac**, seek access:

```
mask_obj:ab
user_obj:abc
user:janea:abdef
foreign_user:/.../cella/fritzb:abc
group:projectx:abcf
group:projecty:bcg
```

Note: The numbered lists in the discussions that follow correspond to stages 1, 2, 3, 4, 5 and 6 of the access-check algorithm referred to in "Access Checking" on page 528.

The principal **janea** requests permission **c**to the object protected by the ACL. Assume that the principal **janea** has the privilege attributes of being a member of the groups**projectx** and **projecty** (as well as having a **user** entry that names her) and that **janea** is the principal to which the **user_obj** entry refers. Assume also that this principal's privilege attributes are certified:

1. The **user_obj** check yields the permissions **abc**.

The result of this check is that the effective permission set for **janea** is **abc**. Because a matching entry is found during the first stage of access checking, none of the remaining stages of access checking is executed, even though there are three other matching entries. The **mask_obj**entry does not mask the **user_obj** entry, so **janea**'s effective permissions are the permissions in the **user_obj** entry. Since **janea** requested a permission that is a member of the effective permission set, her request is granted.

The second principal seeking access to the protected object is **/.../cella/fritzb**. This principal requests permission **b**. Assume that **user_obj** resolves to some identity other than **/.../cella/fritzb,** and that this principal's privilege attributes are uncertified:

1. The **user_obj** check yields no permissions because **/.../cella/fritzb**'s identity does not match that of the **user_obj** (no foreign principal can ever match this entry).
2. The **foreign_user** entry for **/.../cella/fritzb** specifies the permissions **abc**. The application of the **mask_obj**, which specifies the permissions **ab** to this permission set, yields the permissions **ab**. Since the **unauthenticated** mask entry is missing from the ACL, all permissions for unauthenticated identities are masked, yielding an empty effective permission set.

The result of these checks is that **/.../cella/fritzb**'s request is denied (and would be denied, regardless of the permission requested). In this case, only the first two stages of access checking are executed.

The third principal seeking access is **mariac**, who requests permission **a**. Assume that the privilege attributes of **mariac** are certified, that **mariac** is not the principal that corresponds to the **user_obj** entry, and that **mariac** is a member of the groups **projectx** and **projecty**:

1. The **user_obj** check yields no permissions.
2. There is no matching user entry.
3. The group check finds two matching entries. The permissions associated with **projectx** (**abcf**) when masked by the **mask_obj** entry (**ab**) yield the permissions **ab**. The permissions associated with **projecty** (**bcg**) when masked by the **mask_obj** entry yield the permission **b**. The union of the permission sets **ab** and **b** is the set **ab**.

The effective permission set for **mariac** is **ab** and since the requested permission (**a**) is a member of that set, **mariac**'s request is granted. The remaining stages of access checking are not executed.

Example 2

Following is the ACL for an object to which two principals, **ugob** and **/.../cellb/lolad**, seek access:

```
mask_obj:bcde
unauthenticated:ab
user_obj:abcdef
user:ugob:abcdefg
group:projectz:abh
foreign_other:/.../cellb/:abc
```

Note: The numbered lists in the discussions that follow correspond to stages 1, 2, 3, 4, 5 and 6 of the access check algorithm referred to in "Access Checking" on page 528.

The principal **ugob** requests permission **b.**Assume that **ugob** is not the principal to which the **user_obj** entry refers. Assume also that the privilege attributes of **ugob** include membership in the group **projectz**, in addition to the **user** entry that names him. In this case, the principal has failed to acquire certified privilege attributes:

1. The **user_obj** check yields no permissions.

2. The matching entry among the user entries specifies the permissions **abcdefg.** Applying **mask_obj** (**bcde**) yields the permission set**bcde**. Applying the **unauthenticated** mask (**ab**) to the permission set **bcde** yields the effective permission set**b**.

Since the principal **ugob** requests a permission (**b**) that is a member of the effective permissions set, this principal's request is granted.

A case that illustrates how access is determined for otherwise undifferentiated members of a specified foreign cell is that of the principal **/.../cellb/lolad**, who requests permission **e**. Assume that the privilege attributes of this principal are certified:

1. The principal is foreign, so the **user_obj** check cannot be a match.
2. There are no **foreign_user** entries.
3. There are no **foreign_group** entries.
4. The principal **lolad** is a member of **cellb**, meaning that the privilege attributes match those in the **foreign_other** entry for **cellb.** The permissions specified by the **foreign_other** entry for **cellb** (**abc**) as masked by **mask_obj** (**bcde**) yields the effective permission set **bc**.

The permission requested (**e**) is not a member of the effective permission set (**bc**), so the request is denied.

Example 3

Following is the ACL for an object to which one principal, **silviob** seeks access.

```
unauthenticated:a
user:jeand:abcde
user:denisf:-
group:projectx:abcd
foreign_other:/.../cella:-
foreign_other:/.../cellc:abc
any_other:ab
```

Note: The **user** entry for **denisf** and the **foreign_other** entry for **cella** both specify an empty permission set with the notation **-** (dash), meaning that identities corresponding to these entries are explicitly denied all permissions. Also, the numbered lists in the discussions that follow correspond to stages 1, 2, 3, 4, 5 and 6 of the access-check algorithm referred to in "Access Checking" on page 528.

The principal **silviob** requests permission **a**. Assume that this principal's privileges include membership in the group **projecty** and that they are not certified:

1. There is no **user_obj** entry, so this check can yield no permissions.
2. There is no **user** entry for this principal, so this check yields no permissions.
3. There is no entry for the group **projecty**, so this check yields no permissions.
4. There is no **other_obj** entry, so this check can yield no permissions.
5. The principal is local, so no **foreign_other** entry can be a match; this check yields no permissions.
6. Having failed to match any entry examined in the preceding checks, the principal matches the**any_other** entry, which yields the permission set**ab**. There is no **mask_obj** entry, but there is the **unauthenticated** mask entry, which

specifies the permission set **a**. Applying the **unauthenticated** mask to this privilege attribute entry yields the effective permission **a**.

The permission requested (**a**) is a member of the effective permission set (**a**), so this principal's request is granted.

Name-Based Authorization

The Kerberos authentication service, upon which the DCE shared-secret authentication protocol is based, authenticates the string name representation of a principal. The DCE Security Service converts these string representations to UUIDs, and it is these UUIDs that an ACL manager uses to make authorization decisions. However, since some existing (non-DCE) applications implement Kerberos authentication, DCE security supports an authorization protocol based on principal string names: name-based authorization.

It is assumed that applications that use name-based authorization have a means to associate string names with permissions, since the DCE Security Service offers no such facility. Because in name-based authorization there is no UUID representation of privilege attribute data, and because DCE ACL managers recognize only UUIDs, if an application uses name-based authorization, then a principal's privilege attributes are represented as an anonymous PAC. Such PAC data can only match the ACL entry types **other_obj**, **foreign_other**, or **any_other**, and are masked by the **unauthenticated** mask.

Also note that there is essentially no intercell security for an application that uses the name-based authorization protocol because such applications never communicate with the privilege service, which evaluates intercell trust.

Chapter 26. GSSAPI Credentials

A GSSAPI *credential* is a data structure that provides proof of an application's claim to a principal name. An application uses a credential to establish its global identity. The global identity can be, but is not necessarily, related to the local user name under which the application (either the initiator or the acceptor) is running.

A credential can consist of either of the following:

- DCE login context
- Principal name

There are three types of credentials, as shown in Table 18.

Table 18. Credential Types

Credential	Content
INITIATE	A login context only. This credential identifies applications that only initiate security contexts.
ACCEPT	Principal name and an associated entry key table. This credential identifies applications that only accept security contexts.
BOTH	A login context and principal name with a key table entry. This credential identifies applications that can either initiate or accept security contexts.

Credentials are maintained internally to GSSAPI. When they establish a security context, applications use credential handles to point to the credentials they need.

When an application initiates or accepts a security context, it can use GSSAPI routines with either a default credential or a specific credential handle. This chapter discusses how applications do the following:

- Use default credentials
- Create credential handles to refer to specific credentials
- Delegate credentials

For detailed information on the GSSAPI routines referred to in this chapter, see the *OSF DCE Application Development Reference*.

Using Default Credentials

A *default credential* is a credential that is

- Generated by either of the following routines:
  - **gss_init_sec_context()**
  - **gss_accept_sec_context()**
- Based on the following information:
  - The DCE default login context for the application (for **INITIATE** type credentials)
  - The registered principal name in the token (for **ACCEPT** or **BOTH** type credentials).

When an application calls the GSSAPI routine to either initiate (**gss_init_sec_context()**) or accept (**gss_accept_sec_context()**) a security context, it can specify the use of its default credential.

Use default credentials to help ensure the portability of your applications.

Initiating a Security Context

To use a default credential when initiating a security context, an application calls the **gss_init_sec_context()** routine and specifies **GSS_C_NO_CREDENTIAL** as the input claimant credential handle to the routine. The routine uses the initiator's DCE default login context to generate the default credential. The credential is an **INITIATE** type credential.

You can change the default login context by calling the DCE **sec_login_ *()** routines. For information on these routines, see see the appropriate **sec_login_ *(3sec)** reference page.

Accepting a Security Context

To use a default credential when accepting a security context, an application calls the **gss_accept_security_context()** routine and specifies **GSS_C_NO_CREDENTIAL** as the verifier credential handle to the routine. The GSSAPI uses a principal name registered for the context acceptor to generate the default credential handle. The credential is an **ACCEPT** credential type.

Creating New Credential Handles

An application can create a new credential handle to pass to the **gss_init_sec_context()** routine or the **gss_accept_sec_context()** routine. An application might create a credential handle rather than use the default credential for the following reasons:

- Limit the identities the application can use
- Provide an additional identity for the application

Initiating a Security Context with New Credential Handles

To create a credential handle for an **INITIATE** credential type, the application calls the **gssdce_login_context_to_cred()** routine and specifies its login context as input to the routine. The routine creates a credential handle that points to the credential consisting of that login context.

An application can also use a **BOTH** type credential to initiate a security context. Use the **gss_acquire_cred()** routine to create a **BOTH** type credential, as explained in the next section.

When the application uses a **BOTH** credential, the **gss_acquire_cred()** routine creates a login context from the key table information. Then, it uses the login context to create the credential. For more details, see the **gss_acquire_cred(3sec)** reference page.

Accepting a Security Context Using New Credential Handles

To create new credential handle for an **ACCEPT** or **BOTH** type credential, an application calls the **gss_acquire_cred()** routine.

The **gss_acquire_cred()** routine uses a principal name and its entry in the key table to generate the credential handle. If the principal name has not yet been registered (using **gssdce_register_acceptor_identity()** or the **rpc_server_register_auth_info()** routines), the **gss_acquire_cred()** routine automatically registers it.

Delegating Credentials

In delegation, an initiator forwards its identity to an acceptor so that the acceptor can use the identity to act as an agent for the initiator. There are two forms of delegation:

- Impersonation delegation
- Traced delegation

Initiating a Security Context to Delegate Credentials

An application indicates that it wants to delegate credentials when it calls the **gss_init_sec_context()** routine and sets the **GSS_C_DELEG_FLAG** flag to TRUE. Notes added to the initiator's login context can indicate the type of delegation used and any restrictions in effect (for traced delegation only). If no delegation notes are included with the login context and the **GSS_C_DELEG_FLAG** flag is set, impersonation delegation is used.

Accepting a Security Context with Delegated Credentials

If the **GSS_C_DELEG_FLAG** flag has been set when the security context was intiated, the **gss_accept_sec_context()** routine will pass a credential to the acceptor. The routine does the following:

1. Uses information from the input token to create the appropriate delegated credential
2. Creates an impersonation or traced delegation credential with an **INITIATE** credential type
3. Passes the delegated **INITIATE** credential to the acceptor

The principal named in the delegated **INITIATE** credential is the name of the initiator (for impersonation delegation) or the acceptor acting *for* the initiator (for traced delegation). The acceptor uses the credential to act for the initiator, initiating security contexts as appropriate.

Chapter 27. The Extended Privilege Attribute API

This chapter describes the extended privilege attribute (EPA) API. The EPA facility addresses the requirements of complex distributed systems by allowing clients and servers to invoke secure operations via one or more intermediate servers.

In a simple client/server distributed environment, most operations involve two principals: the initiator of the operation and the target of the operation. The target of the operation makes authorization decisions based on the identity of the initiator. However, in distributed object-oriented environments, there is frequently a need for server principals to perform operations on behalf of a client principal. In these cases, it may not be enough for authorization decisions to be based simply on the identity of the initiator since the initiator of the operation may not be the principal that requests the operation.

To handle these cases, the EPA API provides routines that allow principals to operate on objects on behalf of (as delegates of) an initiating principal. The collection of the delegation initiator and the intermediaries is referred to as a delegation chain.

Using the EPA API and related **sec_login_ *()** calls, an application may be written that allows client Principal A to invoke an operation on server Principal C via server Principal B. The DCE Security Service will know the true initiator of the operation (Principal A) and can distinguish the delegated operation from the same operation invoked directly by Principal A.

The EPA interface consists of the security credential calls (**sec_cred_ *()**) that extract privilege attributes and authorization data from an opaque binding handle to authenticated credentials. In addition, the following **sec_login_ *()** calls of the login API are used to establish delegation chains and to perform other delegation related functions.

- **sec_login_become_initiator()**
- **sec_login_become_delegate()**
- **sec_login_become_impersonator()**
- **sec_login_cred_get_delegate()**
- **sec_login_cred_get_initiator()**
- **sec_login_cred_initialize_cursor()**
- **sec_login_disable_delegation()**
- **sec_login_set_extended_attrs()**

Identities of Principals in Delegation

The identity of principals in a delegation chain is maintained in extended privilege attribute certificates (EPACs), as are the identities for all DCE principals. Each EPAC contains the name and group memberships of a principal in the delegation chain and any extended attributes that apply to the principal. The delegation chain includes an EPAC for each member of the delegation chain.

When delegation is in use, the target server receives the delegation chain, and thus knows the privilege attributes of the delegation chain initiator and each intermediary (delegate) in the chain. Authorization decisions can then be made based on the identities of all principals involved in the operation.

ACL Entry Types for Delegation

When a server's ACL manager is presented with credentials to use as a base of an authorization decision, the manager evaluates the privilege attributes of each principal involved in the delegation chain. The ACL manager grants access for the requested operation only if all principals in the delegation chain have the necessary permissions on the object that is the eventual target of the operation.

For the initiator of the delegation chain, permission on the target object must be granted directly using any of the following standard ACL entry types:

- **user_obj**
- **user**
- **foreign_user**
- **group_obj**
- **group**
- **foreign_group**
- **foreign_other**
- **other_obj**
- **foreign_other**
- **any_other**
- **extended**

For intermediaries in a delegation chain, permissions to a target object can be granted directly to the intermediary with the standard ACL entry type previously described, or permissions can be granted by delegate ACL entries. Delegate ACL entries grant permissions to principals only if they are acting as delegates. The following delegate ACL entry types are available:

- **user_obj_delegate**
- **user_delegate**
- **foreign_user_delegate**
- **group_obj_delegate**
- **group_delegate**
- **foreign_group_delegate**
- **foreign_other_delegate**
- **other_obj_delegate**
- **foreign_other_delegate**
- **any_other_delegate**

Note that, to perform an operation, all delegates in the chain must have the appropriate permissions. For example, assume a delegation chain consists of Principal A (the initiator) and Principal's B and C (the intermediaries). To perform the operation, the delegation chain requires **Mrw** permissions on Server X. One way of granting these permission is to grant them directly to each member of the delegation chain, as shown in the following:

```
user:Principal A:Mrw
user:Principal B:Mrw
user:Principal C:Mrw
```

Providing access directly also allows each intermediary in the chain to perform the operation of their own initiative, a consequence that may or may not be desired. To specify that Principals B and C may only be intermediaries operating on behalf of an authorized initiating principal without granting them the ability to perform the operation on their own, use delegation entries. In this case, the Server X's ACL would contain the following entries:

```
user:Principal A:Mrw
user_delegate:Principal B:Mrw
user_delegate:Principal C:Mrw
```

ACL Checking for Delegation

To determine permissions, the ACL manager first uses the standard access-check algorithm (described in "Chapter 25. Authorization" on page 523) to determine the permissions to grant to the delegation initiator. If the requested permission is not granted, access is denied.

If the requested permission is granted, the ACL manager then checks the permissions granted to the delegates in the chain. This checking is similar to the standard access-check algorithm, but it takes into account any additional delegate permissions granted to the delegates. If the requested permission is not granted to all delegates, access is denied. If the requested permission is granted to all delegates, access is granted.

Calls to Establish Delegation Chains

The following **sec_login_ *()** API calls set up a delegation chain:

- **sec_login_become_initiator()**

  Enables delegation for a client. The principal that executes this call is known as the delegation initiator.
- **sec_login_become_delegate()**, **sec_login_become_impersonator()**

  Cause an intermediate server to become a delegate in a delegation chain. The principals that execute these calls are known as intermediaries in the delegation chain.

The **sec_login_become_delegate()** call should be used if the traced delegation has been enabled. The **sec_login_become_impersonator()** call should be used if simple delegation has been enabled. See "Types of Delegation" for more information about delegation types.

The following subsections describe the information supplied to the calls that establish delegation chains.

Types of Delegation

When a client application calls **sec_login_become_initiator()** to enable delegation, that application specifies the type of delegation that should be enabled. The delegation type can be any of the following:

- Traced Delegation

  Includes the identities of all members of the delegation chain in the credentials used for authorization. To become an intermediary in a traced delegation chain, server principals use the **sec_login_become_delegate()** call.

Note that ACLs on objects that are targets of traced delegation must grant the requested permission (or delegate permission) to each member of the delegation chain.

- Impersonation

  Includes only the identity of the initiator of the delegation chain used for authorization. All intermediaries “impersonate” the delegation initiator. To become an impersonator, principals use the **sec_login_become_impersonator()** call.

  Note that ACLs on objects that are targets of impersonation need list only the delegation initiator, not each delegate in the chain.

Generally, traced delegation is the preferred method. The high degree of location transparency inherent in simple delegation greatly increases the risk of a client being compromised by a Trojan horse application.

When server principals run the **sec_login_become_delegate()** or **sec_login_become_impersonator()** call to become an intermediary in a delegation chain, they must also specify the delegation type as input to the call. The type they specify must be the same type as the delegation type specified by the initiator of the chain (unless they specify no delegation).

Target and Delegate Restrictions

When a principal enables delegation or becomes an intermediary in a delegation chain, the principal may specify target and delegate restrictions. Target restrictions identify the server principals (by UUID) to which the identities in a delegation chain can be projected. Delegate restrictions identify the server principals that can further project the delegation chain.

If a target restriction prohibits a server from seeing an identity in a delegation chain, the security runtime replaces that identity with the identity of the anonymous principal. If a delegate restriction prohibits a principal from being an intermediary in a chain, then the security runtime replaces that principal’s identity with the identity of the anonymous principal. This replacement with the anonymous identity allows the authenticated RPC call to complete. Whether the operation requested by the delegation chain is performed can be controlled by ACL entries that grant permission to the anonymous principal on the objects that are the targets of the delegated operation.

If no delegate restrictions are supplied, any principal can be an intermediary in the delegation chain. If any delegate restrictions are supplied, then only those supplied can further transmit the delegation chain.

Note: In the current release of DCE, there is no way for a server to register its DCE credentials with the RPC runtime. Only a server name and key table can currently be registered. Because of this limitation, target restrictions are currently implemented so that *all* target servers see anonymous credentials for any EPAC that contains any target restriction regardless of the identity specified in the restriction.

The Anonymous Principal

The DCE Security Service replaces those identities in the delegation chain that are not allowed to be seen by target or delegate restrictions with the UUIDs associated with the anonymous principal’s identity. These UUIDs are as follows:

- Anonymous principal UUID: **fad18d52-ac83-11cc-b72d-0800092784e9**

- Anonymous group UUID: **fc6ed07a-ac83-11cc-97af-0800092784e9**

The **other_obj**, **any_other**,**other_obj_deleg**, and **any_other_deleg** ACL entries define the anonymous principal's access to objects. The entries must be set up just as for any other principal. The appropriate direct or delegate permissions must be granted to the anonymous principal or the delegated operation will fail.

Target and Delegate Restriction Syntax

Target and delegate restrictions are expressed as a list of values of type **sec_id_restriction_t**. This data type consists of a UUID and an entry type. The entry type specifies whether the UUID identifies a principal, a group, or "any other" principals (in a manner similar to the **any_other** ACL entry type). As in ACL entry types, the target restriction entry types can refer to principals and groups from the local cell or from foreign cells.

The possible delegation entry types are as follows:

- **sec_rstr_e_type_user**

  The target or delegate is a local principal identified by UUID.
- **sec_rstr_e_type_group**

  The target or delegate is any member of a local group identified by UUID.
- **sec_rstr_e_type_foreign_user**

  The target or delegate is a foreign principal identified by principal and cell UUID.
- **sec_rstr_e_type_foreign_group**

  The target or delegate is any member of a foreign group identified by group and cell UUID.
- **sec_rstr_e_type_foreign_other**

  The target or delegate is any principal that can authenticate to the foreign cell identified by UUID.
- **sec_rstr_e_type_any_other**

  The target or delegate is any principal that can authenticate to any cell.
- **sec_rstr_e_type_no_other**

  No principal can act as a target or delegate.

Optional and Required Restrictions

When a principal calls **sec_login_become_initiator()** to enable delegation, or **sec_login_become_delegate()** or **sec_login_become_impersonator()** to become an intermediary, the principal can specify optional and required restrictions. Optional and required restrictions are provided for use by applications that have specific authorization requirements. These restrictions, which are defined by the application, can be set by initiators or intermediaries, and are interpreted and enforced by application target servers. Servers can ignore optional restrictions that they cannot interpret, but they must reject requests associated with a required restriction that they cannot interpret. Both optional and required restrictions are supplied as values of type **sec_id_opt_req_t**. They are inserted in an EPAC by the privilege server and evaluated by the target server application.

Compatibility Between Version 1.1 and Pre-Version 1.1 Servers and Clients

Prior to DCE Version 1.1, a principal's privilege attributes were stored in a privilege attribute certificate (PAC). At Version 1.1, the PAC was renamed to EPAC and extended to include the following:

- Target, delegate, optional, and required restrictions.
- Extended registry attributes (ERAs), as described in "Chapter 29. The Extended Attribute API" on page 551.

Additionally, authorization credentials can now consist of multiple EPACs, as in delegation chains, instead of a single PAC.

When a pre-Version 1.1 client interacts with a Version 1.1 server or vice versa, the Version 1.1 server requires an EPAC and the pre-Version 1.1 server requires a PAC.

For Version 1.1 servers, the security runtime automatically converts the PAC supplied by a pre-Version 1.1 client to an EPAC. For pre-Version 1.1 servers, the security runtime automatically extracts PAC data from the credentials supplied by the Version 1.1 client. However, because an EPAC for a delegation chain contains the privilege attributes of multiple principals and a PAC contains only one set of privilege attributes, the principals engaged in delegation must specify how to handle this issue of multiple versus single identities.

When a principal initiates delegation or becomes an intermediary in a delegation chain, that principal can specify whether to use the privilege attributes of the chain initiator or the last intermediary in the chain to construct the PAC required by a pre-Version 1.1 server. This compatibility decision is specified as a value of type **sec_id_compatibility_mode_t**, which is set to one of the following three values:

- **sec_id_compat_mode_none**

  Compatibility mode is off. The security runtime supplies the application server with an unauthenticated PAC.
- **sec_id_compat_mode_initiator**

  Compatibility mode is on. The pre-Version 1.1 PAC data is extracted from the EPAC of the delegation initiator.
- **sec_id_compat_mode_caller**

  Compatibility mode is on. The pre-Version 1.1 PAC data extracted from the EPAC of the last intermediary in the delegation chain.

Calls to Extract Privilege Attribute Information

The EPA API **sec_cred_ *()** and login API **sec_login_cred_ *()** calls extract privilege attribute information. These calls return information associated with an opaque handle to an authenticated identity.

The **sec_cred_ *()** calls are used by servers that have been called by a client with authenticated credentials. The calls and the information they return are as follows:

- **sec_cred_get_authz_session_info()**

  Returns a client's authorization information
- **sec_cred_get_client_princ_name()**

Returns the principal name of the client

- **sec_cred_get_deleg_restrictions()**

  Returns delegate restrictions
- **sec_cred_get_delegate()**

  Returns a credential handle to the privilege attributes of a delegate in a delegation chain
- **sec_cred_get_delegation_type()**

  Returns the delegation type
- **sec_cred_get_extended_attrs()**

  Returns extended attributes
- **sec_cred_get_initiator()**

  Returns a credential handle to the privilege attributes of the initiator of a delegation chain
- **sec_cred_get_opt_restrictions()**

  Returns optional restrictions
- **sec_cred_get_pa_data()**

  Returns privilege attributes from a credential handle
- **sec_cred_get_req_restrictions()**

  Returns required restrictions
- **sec_cred_get_tgt_restrictions()**

  Returns target restrictions
- **sec_cred_get_v1_pac()**

  Returns pre-Version 1.1 privilege attributes
- **sec_cred_is_authenticated()**

  Returns TRUE if the caller's privilege attributes are authenticated or FALSE otherwise

The **sec_login_cred_ *()** calls are used by clients that are part of a delegation chain. The calls and the information they return are as follows:

- **sec_login_cred_get_delegate()**

  Returns the privilege attributes of a delegate in a delegation chain.
- **sec_login_cred_get_initiator()**

  Returns the privilege attributes of the initiator of a delegation chain

The **sec_cred_ *()** and **sec_login_ *()** calls discussed in this chapter return information about authenticated principals associated with an opaque credential handle supplied to the call. Two credential handles are used:

- **sec_login_handle_t** (returned by a client-side **sec_login_get_current_context()** call)
- **rpc_authz_cred_handle_t** (returned by a server-side **rpc_inq_auth_caller()** call)

These are handles to all the credentials in a delegation chain. The **sec_login_cred_get_initiator()**, **sec_login_cred_get_delegate()**, **sec_cred_get_initiator()**, and **sec_cred_get_delegate()** calls return a handle of type **sec_cred_pa_handle_t**, which is a handle to the extended privilege attributes of a particular identity in a delegation chain. The other **sec_cred_ *()** and **sec_login_ *()** calls discussed in this chapter take the **sec_cred_pa_handle_t** handle and return the requested information for the particular identity.

Disabling Delegation

The login API **sec_login_disable_delegation()** call disables delegation for a specified login context. It returns a new login context of type **sec_login_handle_t** without any delegation information and prevents any further delegation.

Setting Extended Attributes

The login API **sec_login_set_extended_attrs()** call adds extended registry attributes to a login context. The extended registry attributes must have been established and attached to the object by using the ERA API. (For more information on ERAs, see “Chapter 29. The Extended Attribute API” on page 551.)

Chapter 28. The Registry API

This chapter describes the registry API. Like the other security APIs, this one provides a simpler binding mechanism than the standard RPC handle structure. It includes facilities for creating and maintaining the registry database. Applications that run in the default DCE registry environment (that is, those that assume the presence of the default registry tools and servers) have no reason to call this API.

Binding to a Registry Site

Although it is often convenient to speak of the registry database in a way that implies that it is a single physical database, the registry database is replicated in all but the very smallest cells. Replication reduces network traffic and increases the availability of registry data to clients.

A cell's registry database usually consists of an update site (also known as the *master* site), and a number of query sites (also known as read-only, or slave sites). Changes to data at the master site are propagated to its slaves by messages sent by the master. Query sites can only satisfy requests for data (for example, **sec_rgy_acct_lookup()**, which returns account information). Requests for database changes (for example, **sec_rgy_acct_passwd()**, which changes the password for an account) must be directed to the master site; a query site that receives such a request returns an error.

To submit requests to the registry server, a client must first select a site and bind to it. The client may select a site by name, ask the DCE Directory Service to bind to the master site, or select an arbitrary site. In addition, a client may select a cell and bind to a registry site in that cell.

The registry API enables a client to communicate with the registry server via a specified authentication protocol, at a specified protection level, and using a specified authorization protocol. For instance, a developer may decide that the protection level for communicating with an update site should be higher (that is, more secure) than that for a query site; that is, the developer may feel that, on the one hand, the relatively infrequent changes to registry data should be done in a highly secure manner and that, on the other hand, authentication overhead should be reduced for the more frequent requests for registry data. The registry API accommodates these varying needs.

The following calls bind a client to a registry server in preparation for registry operations. The argument list of these calls enables an application to specify the authentication protocol, the protection level, and the authorization protocol to be used:

- **sec_rgy_site_bind()**

  Binds to a specified site
- **sec_rgy_site_bind_update()**

  Binds to the update site
- **sec_rgy_site_bind_query()**

  Binds to any query site
- **sec_rgy_cell_bind()**

  Binds to any registry site in a specified cell
- **sec_rgy_site_binding_get_info()**

Extracts the registry site name and security information from the binding handle

The following calls are similar to the binding calls just described, except that an application cannot specify security information. By default, however, the following calls use DCE shared-secret authentication, the packet-integrity level of protection, and DCE authorization.

- **sec_rgy_site_open()**

  Binds to the specified site
- **sec_rgy_site_open_update()**

  Binds to any update site
- **sec_rgy_site_open_query()**

  Binds to any query site
- **sec_rgy_site_get()**

  Gets the registry site name from the binding handle

The following calls provide miscellaneous binding management functionality:

- **sec_rgy_site_close()**

  Terminates binding to a registry site and frees resources associated with this binding
- **sec_rgy_site_is_readonly()**

  Tests whether a bound site is an update or query site

The Registry Database

The registry database comprises three container objects:

- **principal**

  Contains principal names; each name is associated with account information that is also specified here (for example, the name of the primary group)
- **group**

  Contains groups and the names of their member principals
- **organization**

  Contains organizations and the names of their member principals

These three objects are referred to as name domains, and each member of a domain is referred to as a PGO item. Principal items are contained in the principal domain, groups in the group domain, and organizations in the organization domain. A principal may have a name such as**/rd/writers/tom**, from which you might infer that **tom** is a member of the group **writers** and the organization **rd**. However, this is not the case because the name **/rd/writers/tom** only indicates that **tom** and the data corresponding to the account of this principal (if any) reside in **/rd/writers** in the principal domain. There may also be a group named **/rd/writers** in the group domain, but the principal **tom** is not a member unless he is explicitly named in the group **/rd/writers** in the group domain.

Each PGO item consists of a print string name, a UUID, and a UNIX number (for compatibility with UNIX system security interfaces). For various administrative reasons, it is frequently convenient to be able to refer to a PGO item by more than one name. Consequently, some PGO items are aliases for other items. An alias uses the same UUID and UNIX number as the PGO item to which it refers, but contains only a pointer to that item.

The registry also contains the **rgy** object, which describes registry properties and policies, and organization policies.

Creating and Maintaining PGO Items

The PGO items in the registry database are created and maintained with routines that are prefixed with **sec_rgy_pgo_**. The contents of a PGO item vary with the domain. If the domain is **group** or **organization**, the contents are the membership list of principal names. If the domain is **principal**, the contents are the data corresponding to the registry account using that name.

The **sec_rgy_pgo_ *()** interface contains the following calls for maintaining the PGO trees:

- **sec_rgy_pgo_add()**
  Adds a PGO item
- **sec_rgy_pgo_delete()**
  Deletes a PGO item
- **sec_rgy_pgo_rename()**
  Changes the name of a PGO item
- **sec_rgy_pgo_replace()**
  Replaces information corresponding to the specified PGO item

The **sec_rgy_pgo_ *()** interface contains the following calls for maintaining PGO membership lists:

- **sec_rgy_pgo_add_member()**
  Adds a member to a group or organization membership list
- **sec_rgy_pgo_delete_member()**
  Deletes a member from a group or organization membership list
- **sec_rgy_pgo_get_members()**
  Returns a list of members of a group or organization
- **sec_rgy_pgo_is_member()**
  Tests whether a principal is a member of a specified group or organization

The **sec_rgy_pgo_ *()** interface contains the following calls for retrieving PGO item data:

- **sec_rgy_pgo_get_by_id()**
  Returns the PGO item with the specified UUID
- **sec_rgy_pgo_get_by_eff_unix_num()**
  Returns the PGO item with the specified effective UNIX number
- **sec_rgy_pgo_get_by_name()**
  Returns the PGO item with the specified name
- **sec_rgy_pgo_get_by_unix_num()**
  Returns the PGO item with the specified UNIX number
- **sec_rgy_pgo_get_next()**
  Returns the PGO item that follows the last PGO item returned

The **sec_rgy_pgo_ *()** interface also contains routines that convert PGO item specifiers, as follows:

- **sec_rgy_pgo_id_to_name()**

- **sec_rgy_pgo_id_to_unix_num()**
- **sec_rgy_pgo_name_to_id()**
- **sec_rgy_pgo_unix_num_to_id()**
- **sec_rgy_pgo_name_to_unix_num()**
- **sec_rgy_pgo_unix_num_to_name()**

Creating and Maintaining Accounts

The *login-name* field of an account contains a principal name, a primary group name, and an organization name. The account may also contain a project list (also known as a concurrent group set) that specifies all the groups to which the principal corresponding to the account belongs, but the *login-name* field itself specifies only one group name.

An account can be added to the registry database only when all of its constituent PGO items are established. For instance, to create an account with the principal name **tom**, the group name **writers**, and the organization name **rd**, all three names must exist as individual PGO items in the database; and the **writers** group and the **rd** organization must specify that **tom** is a member.

When an account is created with **sec_rgy_acct_add()** (and if a project list is enabled for the new account), the call scans the groups in the registry and creates a project list containing all the groups in which the principal name appears. Subsequently, the project list may be modified with the **sec_rgy_pgo_add_member()** and **sec_rgy_pgo_delete_member()** calls.

The following calls create and maintain accounts:
- **sec_rgy_acct_add()**
  Adds an account to an existing principal item
- **sec_rgy_acct_delete()**
  Deletes an account, leaving the principal item
- **sec_rgy_acct_rename()**
  Changes an account login name, perhaps moving the account to a different principal item

The following calls return the information in an account:
- **sec_rgy_acct_get_projlist()**
  Returns the project list for an account
- **sec_rgy_acct_lookup()**
  Returns all the account data

The following calls modify the information in an account:
- **sec_rgy_acct_passwd()**
  Changes an account password
- **sec_rgy_acct_replace_all()**
  Replaces all of an account's data
- **sec_rgy_acct_admin_replace()**
  Replaces only the administrative account data
- **sec_rgy_acct_user_replace()**
  Replaces only the account data that is accessible to the user of the account

Registry Properties and Policies

The following subsections outline some registry API parameters that affect the cell as a whole, and the routines that enable an application to retrieve and set values for them.

Registry Properties

Several registry parameters and flags affect all accounts in the registry. These registry properties include the following:

- The version number of the registry software used to create and read the registry
- The name and UUID of the cell associated with the registry, and whether the current registry site is an update site or a query site
- Minimum and default lifetimes for certificates of identity issued to principals
- Bounds on the UNIX numbers used for principals, and whether the UUIDs of principals also contain embedded UNIX numbers

The routines associated with this parameter set are

- **sec_rgy_properties_get_info()**
- **sec_rgy_properties_set_info()**

The Registry Authentication Policy

Another set of parameters affecting all principals is the registry authentication policy. This set only controls the maximum lifetime of certificates of identity, upon first issue and renewal. Accounts also have authentication policies, and the policy in effect for any principal is the most restrictive combination of the registry policy and the policy for a principal's account. The associated routines are

- **sec_rgy_auth_plcy_get_info()**
- **sec_rgy_auth_plcy_get_effective()**
- **sec_rgy_auth_plcy_set_info()**

Organization Policies

Another parameter set controls the set of accounts of principals that are members of an organization. These parameters control the lifetime and length of passwords, as well as the set of characters from which passwords may be composed. This parameter set also specifies the default lifespan of accounts associated with the organization. The routines associated with this parameter set are

- **sec_rgy_plcy_get_info()**
- **sec_rgy_plcy_get_effective()**
- **sec_rgy_plcy_set_info()**

Routines to Return UNIX Structures

The registry API provides calls to obtain registry entries in a UNIX compatible structure. These APIs return account and group entries similar to the **getpwnam**, **getgrnam**, **getpwuid**, and **getgrid** UNIX library routines. These APIs, which can be called by the corresponding UNIX library routines to ensure compatibility with UNIX programs, are

- **sec_rgy_unix_getpwnam()**

  Returns a UNIX compatible password entry for an account specified by name

- **sec_rgy_unix_getgrnam()**

  Returns a UNIX compatible group entry for an account associated with a specified group name
- **sec_rgy_unix_getpwuid()**

  Returns a UNIX compatible password entry for an account specified by UNIX ID
- **sec_rgy_unix_getgrgid()**

  Returns a UNIX compatible group entry for an account associated with a specified group ID

Miscellaneous Registry Routines

The registry API includes a few miscellaneous routines, as follows:

- **sec_rgy_login_get_info()**

  Returns login information for the specified account.
- **sec_rgy_login_get_effective()**

  Applies local overrides (if such data is available) to registry account information and returns information about which account information fields have been overridden
- **sec_rgy_wait_until_consistent()**

  Blocks until all previous database updates have been propagated to all sites. This is useful for applications that first bind and write to an update site, and then bind to an arbitrary query site and depend upon up-to-date information.

  Note: The **sec_rgy_wait_until_consistent()** routine is not available in DCE Release 1.0 Version 1.0.2.
- **sec_rgy_cursor_reset()**

  Resets the database cursor to return the first suitable entry

Chapter 29. The Extended Attribute API

This chapter describes the extended attribute APIs. There are two extended attribute APIs: the extended registry attribute (ERA) interface to create attributes in the registry database and the DCE attribute interface to create attributes in a database of your choice.

The ERA interface (consisting of **sec_attr_ *()** calls) provides facilities for extending the registry database by creating, maintaining, and viewing attribute types and instances, and providing information to and receiving it from outside attribute servers known as attribute triggers. It is the preferred API for security schema and attribute manipulations. Application servers that manage legacy security attributes or provide third-party processing of attributes stored in the registry database can export and implement the **sec_attr()** interface. Trigger servers are accessed through the **sec_attr_trig()** interface by the security client agent during certain **sec_rgy_attr_ *()** calls. The ERA interface uses the same binding mechanism as the registry API, described in "Chapter 28. The Registry API" on page 545.

The DCE attribute interface (consisting of **dce_attr_sch_ *()** calls) is provided for schema and attribute manipulation of data repositories other than the registry. Although similar to the ERA interface, the functionality of the DCE attribute interface is limited to creating schema entries (attribute types). The interface does not provide calls to create and manipulate attribute instances or to access trigger servers.

The chapter first describes the ERA interface and then the DCE attribute interface. Finally is describes macros and utilities provided for developers who use either attribute API.

The ERA API

The registry is a repository for principal, group, organization, and account data. It stores the network privilege attributes used by DCE and account data used by local operating systems. This local account data, however, is appropriate only for UNIX operating systems. The ERA facility provides a mechanism for extending the registry schema to include data (attributes) required by or useful to operating systems other than UNIX operating systems.

The ERA API provides the ability to define attribute types and to attach attribute instances to registry objects. Registry objects are nodes in the registry database, to which access is controlled by an ACL manager type. The registry objects are

- **principal**
- **group**
- **organization**
- **policy**
- **directory**
- **replist**
- **attr_schema**

All registry objects and their accompanying ACL manager type are described in the *OSF DCE Administration Guide—Core Components*.

The ERA API also provides a trigger interface that application servers use to integrate their attribute services with ERA services.

Attribute Schema

The schema extensions are implemented in a single attribute schema that is essentially a catalog of schema entries, each of which defines the format and function of an attribute type. The schema can be dynamically updated to create, modify, or delete schema entries.

The attribute schema is identified by the name **xattrschema** under the security junction point (usually **/.:/sec**) in the CDS namespace. Access to the attribute schema (hereinafter called simply schema) is controlled by an ACL on the schema object. The schema is propagated from the master security server to replicas, like other registry data. Since the attribute schema is local to a cell, it defines the types that can be used within the cell, but not outside the cell (unless the type is also defined in another cell).

Attribute Types and Instances

Each attribute type definition in the schema consists of attribute type identifiers (UUID and name) and semantics that control the instances of attributes of this type. In this manual, schema entry refers to the registry entry that defines an attribute type.

An attribute instance is an attribute that is attached to an object and has a value (as opposed to an attribute type, which has no values but simply defines the semantics to which attribute instances of that attribute type must adhere). Attribute instances contain the UUID of their attribute type.

Attribute Type Components

The **sec_attr_schema_entry_t** data type defines an attribute type. This data type contains attribute type identifiers and characteristics.

The identifiers of attribute types are a name and a UUID. Generally, the name is used for interactive access and the UUID for programmatic access.

Attribute type characteristics describe the format and function of the attribute type and thus control the format and function of instances of that type. These characteristics, all specified in the **sec_attr_schema_entry_t** data type, are described in the following sections.

Attribute Encoding

Attribute encoding defines the legal encoding for instances of the attribute type. The encoding controls the format of the attribute instance values, such as whether the attribute value is an integer, string, a UUID, or a vector of UUIDs that define an attribute set.

Attribute encodings are specified in the **sec_attr_encoding_t** data type (fully described in the *OSF DCE Application Development Reference*).

The possible encodings for attribute types are

- **any**

The attribute instance value can be of any legal encoding type.

- **void**

  The attribute instance has no value. It is simply a marker that is either present or absent.
- **printstring**

  The attribute value is a printable IDL character string from the DCE Portable Character Set (PCS).
- **printstring_array**

  The attribute value is an array of print strings.
- **integer**

  The attribute value is a signed 32-bit integer.
- **bytes**

  The attribute value is a string of bytes. The byte string is assumed to be a pickle or is otherwise a self-describing type.
- **confidential_bytes**

  The attribute value is a string of encrypted bytes. This encrypted data can be passed over the network and is available to user-developed applications.
- **internationalization_data**

  An internationalized string of bytes with a tag identifying the OSF registered codeset used to encode the data.
- **uuid**

  A DCE UUID.
- **attr_set**

  The value is an attribute set, a vector of attribute type UUIDs used to associate multiple related attribute instances (members of the set). The vector contains the UUIDs of each member of the set. Attribute sets provide a flexible way to group related attributes on an object for easier search and retrieval.

  The attribute type UUIDs referenced in an attribute set instance must correspond to existing attribute schema entries. Although the members specified in a set are generally expected to be attached to the object to which the set instance is attached, no checking is done to confirm that they are. Thus, it is possible to create an attribute set instance on an object before creating member attribute instances on that object. A query on such an attribute set returns all instances of member attributes that exist on the object along with a warning that some attribute types were missing.

  Note that attribute sets cannot be nested; a member UUID of an attribute set cannot itself identify an attribute set.

  A query on an attribute set expands to a query per the set’s members. In other words, an attribute lookup operation on an attribute set returns all attribute instances that are members of the set, not the set instance itself. (Certain operations, such as **sec_rgy_attr_set_lookup_by_id()** and **sec_rgy_attr_lookup_by_name()**, can retrieve attribute set instances.)

  Updates to an attribute set (**sec_rgy_attr_update()**) do not expand the update to its members but apply only to the attribute set. Since the value carried by a set instance is a vector containing the UUIDs of the member attribute types, an update makes changes only to the set’s members, not the values carried by those member attributes. Deletions of attribute sets delete only the set instance, not the member instances.

  Since the attributes that are set members exist independently of the attribute set, they can be manipulated directly like any other attribute.

- **binding**

  The attribute value is a **sec_attr_binding_info_t** type containing authentication, authorization, and binding information suitable for communicating with a DCE server.

ACL Manager Set

An attribute type's ACL manager set specifies the ACL manager type or types (by UUID) that control access to the object types to which attribute instances of this type can be attached. Attribute instances can be attached only to objects protected by the ACL manager types in the schema entry. For example, suppose an ACL manager set for an attribute type named **MVSname** lists only the ACL manager type for principals. Then, instances of the attribute type named **MVSname** can be attached only to principals and not any other registry objects.

Access to an attribute instance is controlled by the ACL on the object to which the attribute instance is attached and access control is implemented by the object's ACL manager type. For example, access to an attribute named **MVSname** on the principal object named **delores** is controlled by the ACL on the **delores** object.

Do not confuse access to an attribute type definition (a schema entry) with access to an attribute instance. As described previously, access to a schema entry is controlled by the ACL on the **xattrschema** object. Access to an attribute instance is controlled by the ACL on the object to which the attribute instance is attached.

In addition to the ACL manager types, the ACL manager set defines the permission bits needed to query, update, test, and delete instances of the attribute type. These bits are used by the object's ACL manager to determine rights to the object's attributes.

The ACL manager types and permissions defined for the attribute type apply to all instances of the attribute type.

Note that the ACL manager facility supports additional generic attribute type permissions (**O** through **Z** inclusive). Administrators can assign these permissions to attribute types of their choice. All uses of these additional permission bits are controlled by the cell's administrator. See the *OSF DCE Administration Guide—Core Components* for more information.

Attribute Flags

The attribute type flags set in a schema entry are described in the following paragraphs.

The Unique Flag: The unique flag specifies whether the value of each instance of an attribute type must be unique within the cell. For example, assume that an instance of attribute type A is attached to 25 principals in the cell. If the unique flag is set on, the value of the A attribute for each of those 25 principals must be different. If it is set off, all 25 principals can share the same value for attribute A.

The Multivalued Flag: The multivalued flag specifies whether instances of the attribute can be multivalued. If an attribute is multivalued, multiple instances of the same attribute type can be attached to a single registry object. For example, if the multivalued flag is set on, a single principal can have multiple instances of attribute type A. If the flag is set off, a single principal can have only one instance of attribute type A.

All instances' multivalued attributes share the UUID (the UUID of their attribute type), but the values carried by the instances differ. Generally, to access all instances of a multivalued attribute, you supply the attribute UUID. To access a specific instance of a multivalued attribute, you supply the UUID and the value carried by that instance.

The Reserved Flag: The reserved flag indicates whether the attribute type can be deleted from the schema. Note that, when an attribute type is deleted, all instances of the attribute type are deleted. If the reserved flag is set on, the entry cannot be deleted. If the reserved flag is set off, authorized principals can delete the schema entry.

The Apply-Defaults Flag: The apply-defaults flag indicates whether or not default attributes should be returned when objects are queried by a client with the **sec_rgy_attr_get_effective()** call. If the apply-defaults flag is set on, defaults are applied. If it is set off, defaults are not supplied.

Defaults are determined in the following manner:

1. If the requested attribute exists on the principal, that attribute is returned. If it does not, the search continues.
2. The next step in the search depends on the type of object:

   For principals with accounts:

   - The organization named in the principal's account is examined to see if an attribute of the requested type exists. If it does, it is returned and the search ends. If it does not, the search continues to the **policy** object as described in Step 2b.
   - The registry **policy** object is examined to see if an attribute of the requested type exists. If it does, it is returned. If it does not, a message indicating that no attribute of the type exists for the object is returned.

   For principals without accounts, for groups, and for organizations:

   The registry **policy** object is examined to see if an attribute of the requested type exists. If it does, it is returned. If it does not, a message indicating that no attribute of the type exists for the object is returned.

The Intercell Action Field

The intercell action field of the schema entry specifies the action that should be taken by the privilege server when reading attributes from a foreign cell. This field can contain one of three values:

- **sec_attr_intercell_act_accept**

  To accept the foreign attribute instance
- **sec_attr_intercell_act_reject**

  To reject the foreign attribute instance
- **sec_attr_intercell_act_evaluate**

  To call a remote trigger server to determine how the attribute instance should be handled

When the privilege server generates a PTGT for a foreign principal, it retrieves the list of attributes from the foreign principal's EPAC.

These attributes instances may be attached to the **principal** object itself or attached to the group or **organization** object associated with the **principal** object.

The privilege server then checks the local attribute schema for attribute types with UUIDs that match the UUIDs of the the attribute instances from the foreign cell that are contained in the EPAC. At this point, the privilege server takes one of the following two actions:

1. If the privilege server cannot find a matching attribute type in the local attribute schema, it checks the **unknown_intercell_action** attribute on the **policy** object. If the **unknown_intercell_action** attribute is set to
   - **sec_attr_intercell_act_accept**, the foreign attribute instance is retained and included in the EPAC generated for the object by the privilege server.
   - **sec_attr_intercell_act_reject**, the foreign attribute is discarded.

   Note: The **unknown_intercell_action** attribute must be created by the system administrator and attached to the **policy** object. The attribute type, which takes the same values as the intercell_action field, has the following characteristics:

   Name: **unknown_intercell_action**

   Attribute UUID: **171e0ef2c-d12e-11cc-bb7b-080009353559**

   Encoding: **sec_attr_encoding_integer**

   ACL manager set: **policy_acl_mgr**

   Unique: false

   Multivalued: false

   Reserved: true

   Comment text: Flag indicating whether to accept or reject foreign attributes for which no schema entry exists
2. If the privilege server finds a matching attribute type in the local attribute schema, it retrieves the attribute. The action it now takes depends on the setting of the attribute type's intercell action field and unique flag as follows:
   - If the intercell action field is set to **sec_attr_intercell_act_accept** and
     – The unique flag is not set on, the privilege server includes the foreign attribute instance in the principal's EPAC.
     – The unique flag is set on, the privilege server includes the foreign attribute instance in the principal's EPAC only if the attribute instance value is unique among all instances of the attribute type within the local cell.

     Note: If the unique attribute type flag is set on and a query trigger exists for a given attribute type, the intercell action field cannot be set to **sec_attr_intercell_act_accept** because, in this case, only the query trigger server can reasonably perform a uniqueness check.
   - If the intercell action field is set to **sec_attr_intercell_act_reject**, the privilege server unconditionally discards the foreign attribute instance.
   - If the intercell action field is set to **sec_attr_intercell_act_evaluate**, the privilege server makes a remote **sec_attr_trig_intercell_avail()** call to an attribute trigger by using the binding information in the local attribute type schema entry. The remote attribute trigger decides whether to retain, discard, or map the attribute instance to another value(s). The privilege server

includes the values returned by the attribute trigger in the **sec_attr_trig_query()** call output array in the principal's EPAC.

Attribute Scope

The scope field controls the objects to which the attribute can be be attached. If scope is defined, the attribute can be attached only to objects defined by the scope. For example, if the scope for a given attribute type is defined as the directory name **/principal/krbgt**, instances of that attribute type can be attached only to objects in the **/principal/krbgt** directory (a directory that by convention contains only cell principals). If the scope is narrowed by fully specifying an object in the **/principal/krbgt** directory (for example, **/principal/krbgt/dresden.com**) then the attribute can be attached only to the **dresden.com** principal.

Trigger Type Flag

The schema entry trigger type flag specifies whether the trigger server associated with the attribute type is invoked for update or query operations. See "The Attribute Trigger Facility" on page 565 for more information on attribute triggers.

Trigger Binding

The schema entry trigger binding field contains a binding handle to a remote trigger that will perform processing for the attribute instances. See "The Attribute Trigger Facility" on page 565 for more information on attribute triggers.

Calls to Manipulate Schema Entries

This section first introduces the **sec_attr_schema_entry_t** data type used by the calls that create and update schema entries that define attribute types. It then describes the calls that create, modify, delete, and read schema entries.

The sec_attr_schema_entry_t Data Type

The **sec_attr_schema_entry_t** data type is used in the calls that create and update schema entries. The data type consists of four values and six other data types. The values used by the **sec_attr_schema_entry_t** are the attribute type name, UUID, scope, and a text field for comments.

The data types used by the **sec_attr_schema_entry_t** are

- **sec_attr_sch_entry_flags_t**

  Specifies the unique, multivalued, reserved, and apply defaults attribute flags.
- s**ec_attr_acl_mgr_info_set_t**

  Specifies the attribute type's ACL manager(s). This data type defines the attribute type ACL manager set. This data type contains an array of pointers of type **sec_attr_mgr_info_p_t**, which reference **sec_attr_acl_mgr_info_t** data types. There is one **sec_attr_acl_mgr_info_t** data type for each ACL manager associated with the attribute type. Each **sec_attr_acl_mgr_info_t** defines the ACL manager UUID and the permission bits.
- **sec_attr_encoding_t**

  Specifies the schema entry encoding.
- **sec_attr_trig_type_t**

Specifies the type of attribute trigger associated with the attribute type (if an attribute trigger is to be associated with the attribute type). See "The Attribute Trigger Facility" on page 565 for more information on attribute triggers.

- **sec_attr_intercell_action_t**

  Specifies the action to be taken attribute instances of this type that come from a foreign cell.
- **sec_attr_bind_info_t**

  Specifies binding information for the trigger server associated with the attribute type (if an attribute trigger is associated with the attribute type).

  The **sec_attr_bind_info_t** data type uses two other data types: **sec_attr_bind_auth_info_t** and **sec_attr_binding_t**. The **sec_attr_bind_info_t** structure for trigger binding is described fully in "The Attribute Trigger Facility" on page 565.

Figure 77 illustrates the structure of a **sec_attr_schema_entry_t** data type.

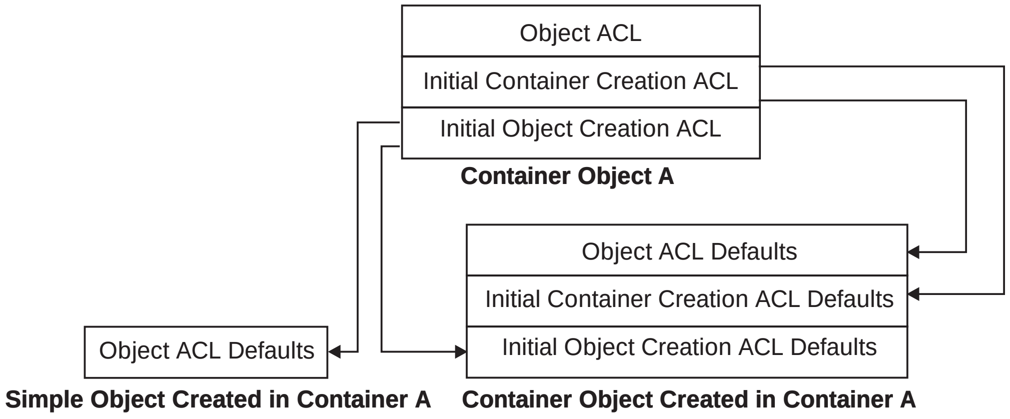

Object ACL
Initial Container Creation ACL
Initial Object Creation ACL
Container Object A
Object ACL Defaults
Initial Container Creation ACL Defaults
Initial Object Creation ACL Defaults
Object ACL Defaults
Simple Object Created in Container A
Container Object Created in Container A

Figure 77. The sec_attr_schema_entry_t Data Type

Creating and Managing Schema Entries

This section describes the calls to create, modify, and delete the schema entries that define attribute types.

The sec_rgy_attr_sch_create_entry() Call

The **sec_rgy_attr_sch_create_entry()** call creates a schema entry that defines an attribute type in the attribute schema.

This call uses the **sec_attr_schema_entry_t** data type that completely defines the schema entry, including the following:

- The attribute type name (generally used for interactive access) and UUID (generally used for programmatic access). Note that attribute instances share the name and UUID of their attribute type.
- The attribute's encoding (described in "Attribute Type Components" on page 552). The encoding is specified as an enumerator of type **sec_attr_encoding_t**. For some kinds of encodings, additional data types are used to further specify the encoding information. These additional data types, the kinds of encodings that require them, and the purpose of the data types are listed in Table 19 on page 559.

Table 19. Encodings and Required Data Types

Encoding	Required Data Type	Purpose of Data Type
sec_attr_enc_bytes	**sec_attr_enc_bytes_t**	Defines the length of attribute values
sec_attr_enc_confidential_bytes	**sec_attr_enc_bytes_t**	Defines the length of attribute values
sec_attr_enc_i18n_data	**sec_attr_i18n_data_t**	Defines the internationalization codeset
sec_attr_enc_attr_set	**sec_attr_enc_attr_set_t**	Defines the total number of members in the attribute set and the UUID of each member
sec_attr_enc_printstring	**sec_attr_enc_printstring_t**	Defines a single print string
sec_attr_enc_printstring_array	**sec_attr_enc_str_array_t**	Defines an array of print strings

The sec_rgy_attr_sch_update_entry() Call

The **sec_rgy_attr_sch_update_entry()** call updates a schema entry that defines an attribute type.

The schema entry components that can be modified are controlled by the ERA API and the *modify_parts* parameter of the **sec_rgy_attr_sch_update_entry()** call.

To ensure that registry and access control data remains consistent, the ERA API allows only the following schema entry components to be modified:

- Attribute name
- Reserved flag
- Apply defaults flag
- Intercell action flag
- Trigger binding
- Comment

Note that ACL managers can be added to a schema entry's ACL manager set, but they cannot be modified or deleted.

To modify any other schema entry fields implies a drastic change to the attribute type. If this change must be made, the schema entry must be deleted (which deletes all attribute instances of that type) and then recreated.

The *modify_parts* parameter of the **sec_rgy_attr_sch_update_entry()** call can also be used to prohibit modification of additional schema entry fields. This parameter, which is actually a **sec_attr_schema_entry_parts_t** data type, lists the fields that can be modified by the call. Only those fields listed in **sec_attr_schema_entry_parts_t** can be modified.

The new values used to update the attribute type are supplied in a **sec_attr_schema_entry_t** data type.

The sec_rgy_attr_sch_delete_entry() Call

The **sec_rgy_attr_sch_delete_entry()** call deletes attributes types from the attribute schema. The attribute type to be deleted is specified by UUID. When an attribute type is deleted, all instances of that attribute type are invalidated.

Reading Schema Entries

This section describes the calls that read schema entries and the cursor used by the **sec_rgy_attr_sch_scan()** call.

Using sec_attr_cursor_t with sec_rgy_attr_sch_scan()

The **sec_rgy_attr_sch_scan()** call, which reads a specified number of attribute type entries from the attribute schema, uses a cursor of type **sec_attr_cursor_t**. This cursor must be allocated before it can be used as input to the **sec_rgy_attr_sch_scan()** call. In addition, it can also be initialized to the first attribute type entry in the schema, although this is not required. After use, the resources allocated to the **sec_attr_cursor_t** must be released.

The following calls allocate, initialize, and release a **sec_attr_cursor_t** for use with the **sec_rgy_attr_sch_scan()** call:

- **sec_rgy_attr_sch_cursor_init()**

  The **sec_rgy_attr_sch_cursor_init()** call allocates resources to the cursor and initializes the cursor to the first attribute type entry in the attribute schema. This call also supplies the total number of entries in the attribute schema as part of its output. The cursor allocation is a local operation. The cursor initialization is a remote operation and makes a remote call to the registry.
- **sec_rgy_attr_sch_cursor_alloc()**

  The **sec_rgy_attr_sch_cursor_alloc()** call allocates resources to the cursor but does not initialize the cursor. However, since the **sec_rgy_attr_sch_scan()** call will initialize the cursor if it is passed in uninitialized, you may prefer this call to limit the number of remote calls performed by an application. Be aware that the **sec_rgy_attr_sch_cursor_init()** call provides the total number of entries in the named schema, a piece of information not provided by the **sec_rgy_attr_sch_cursor_alloc()** call.
- **sec_rgy_attr_sch_cursor_release()**

  The **sec_rgy_attr_sch_cursor_release()** call releases all resources allocated to a **sec_attr_cursor_t** cursor used with the **sec_rgy_attr_sch_scan()** call.
- **sec_rgy_attr_sch_cursor_reset()**

  The **sec_rgy_attr_sch_cursor_reset()** call initializes a **sec_attr_cursor_t** cursor used with the **sec_rgy_attr_sch_scan()** call. The reset cursor can then be used without releasing and reallocating.

The sec_rgy_attr_sch_scan() Call

The **sec_rgy_attr_sch_scan()** call reads a specified number of schema entries from the attribute schema.

The number of entries to read is specified as an unsigned 32-bit integer. The read begins at the entry at which the **sec_attr_cursor_t** cursor is positioned and continues through the number of entries specified. The cursor must be allocated but can be initialized or uninitialized since **sec_rgy_attr_sch_scan()** initializes any uninitialized cursor it receives as input.

The call output includes an array of **sec_attr_schema_entry_t** values and a 32-bit integer that specifies the number of schema entries returned.

To read through all entries in a schema, continue making **sec_rgy_attr_sch_scan()** calls, until the **no_more_entries** message is received. When all calls are complete, release the resources allocated to the **sec_attr_cursor_t** cursor by using the **sec_rgy_attr_sch_cursor_release()** call.

The sec_rgy_attr_sch_lookup_by_id() and sec_rgy_attr_sch_lookup_by_name() Calls

The **sec_rgy_attr_sch_lookup_by_id()** call reads the attribute schema entry identified by UUID. The output of the call is a **sec_attr_schema_entry_t** type that contains the specified attribute type's name, UUID, and characteristics. Generally, this call is used for programmatic access.

For interactive access, use the **sec_rgy_attr_sch_lookup_by_name()** call. This call returns the same information as the **sec_rgy_attr_sch_lookup_by_id()** call but specifies the schema entry to read by name instead of by UUID.

Reading the ACL Manager Types

Two calls retrieve the ACL manager types that protect objects dominated by a named schema:

- **sec_rgy_attr_sch_get_acl_mgrs()**

  Retrieves the UUIDs of the ACL manager types protecting all objects in a named schema.
- **sec_rgy_attr_sch_aclmgr_strings()**

  Retrieves printable strings for each ACL manager type protecting objects in a named schema. The strings contain the ACL manager type's name, associated help information, and supported permission bits.

Calls to Manipulate Attribute Instances

This section introduces the **sec_attr_schema_t** data type used by the calls that create and update attribute instances and then describes the calls that create, modify, delete, and read attribute instances. For all calls, the object whose attributes should be accessed is identified by name and by the domain in which the object exists. (The domain parameter is ignored for the **Policy** and the **Replist** objects.) Registry domains are described in “Chapter 28. The Registry API” on page 545.

The sec_attr_t Data Type

The **sec_attr_t** data type is used in the calls that create and update attribute instances. The data type consists of a value of type **uuid_t** that identifies the attribute to be accessed by UUID and data type of **sec_attr_value_t**. The **sec_attr_value_t** data type is a tagged union of the actual value assigned (or to be assigned to the attribute instance) and a data type of **sec_attr_encoding_t** that specifies the encoding tags that define the attribute type characteristics. Figure 78 on page 562 illustrates the structure of a **sec_attr_t data** type.

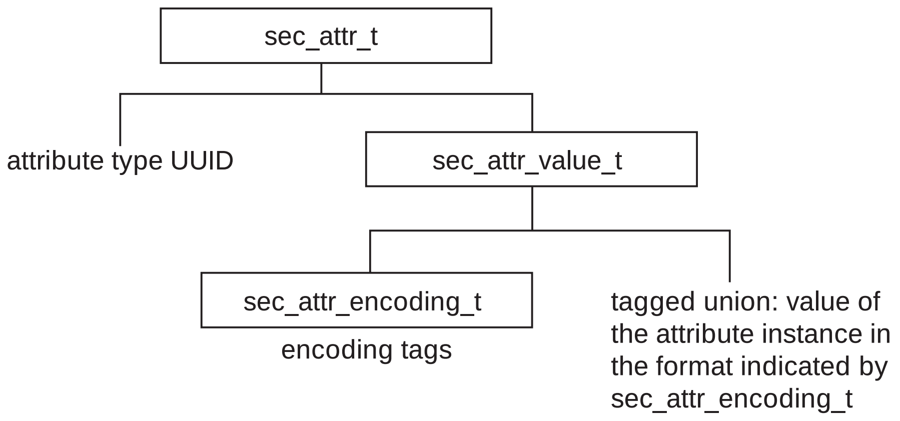

sec_attr_t
attribute type UUID
sec_attr_value_t
sec_attr_encoding_t
encoding tags
tagged union: value of the attribute instance in the format indicated by sec_attr_encoding_t

Figure 78. The sec_attr_t Data Type

Creating and Managing Attribute Instances

This section describes the calls to create, modify, and delete the attribute instances.

The sec_rgy_attr_update() Call

The **sec_rgy_attr_update()** call creates new attribute instances and updates existing attribute instances attached to an object specified by name and registry domain. The instances to be created or updated are passed as an array of **sec_attr_t** data types.

Because the new values are passed in as an array, if the update of any attribute instance in the array fails, all fail. However, to help pinpoint the cause of the failure, the call identifies the first attribute whose update failed in a failure index by array element number.

For existing attribute instances attached to the object, the values passed in the array overwrite the existing values. In other words, if the UUID passed in the input array matches the UUID of an existing instance, the values passed in overwrite the existing values.

If the attribute instance does not exist, it is created. In other words, if the UUID passed in in the array does not match any other attribute type UUID attached to the object, a new attribute instance is created.

For multivalued attributes, because every instance of the multivalued attribute is identified by the same UUID, every instance is overwritten with the supplied value. For example, suppose object **delores** has three attributes of the multivalued type **security_role**. If you pass in one value for **security_role**, the values of all three are changed to the one you enter.

To change only one of the **security_role** values, you must supply the values that should be unchanged as well as the new value. For example, suppose object **delores** has three **security_role** attributes with values of **level1**, **level2**, and **level3**. To change **level1** to **level1.5 and retainlevel2** and **level3**, the input array must contain **level1.5**, **level2**, and **level3**.

To create instances of multivalued attributes, you must create individual **sec_attr_t** data types to define each multivalued attribute instance and then pass all of them in the **sec_rgy_attr_update()** input array.

If an input attribute is associated with an update attribute trigger, the attribute trigger is invoked (by the **sec_attr_trig_update()** call), and the values in the **sec_rgy_attr_update()** input array are used as input to the update attribute trigger. The output values from the update attribute trigger are stored in the registry database and returned in the **sec_rgy_attr_update()** output array.

The sec_rgy_attr_test_and_update() Call

The **sec_rgy_attr_test_and_update()** call, like the **sec_rgy_attr_update()** call, creates new attribute instances and updates existing attribute instances attached to an object specified by name and registry domain. However, it performs the update only if a set of specified attribute instances match the attribute instances that already exist for the object. This call is useful to ensure that updates are made only if certain conditions exist.

The attribute instances to be matched are passed in an input array of **sec_attr_t** values. Other than this conditional test, this call functions exactly the same as the **sec_rgy_attr_update()** call.

The sec_rgy_attr_delete() Call

The **sec_rgy_attr_delete()** call deletes the specified attribute instances from an object identified by name and registry domain. The attribute instances to be deleted are passed in as an array of values of **sec_attr_t**.

To delete attribute instances that are not multivalued and to delete all instances of a multivalued attribute, an attribute UUID is all that is required. For these attribute instances, supply the attribute UUID in the input array and set the attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void**.

To delete a specific instance of a multivalued attribute, you must supply the UUID and value that uniquely identify the multivalued attribute instance in the input array.

Note that, if the deletion of any attribute instance in the array fails, all fail. However, to help pinpoint the cause of the failure, the call identifies the first attribute whose deletion failed in a failure index by array element number.

Reading Attribute Instances

This section describes the calls that read attribute instances, and it describes the cursor used by the **sec_rgy_attr_lookup_by_id()** call.

Using sec_rgy_attr_cursor_t with sec_rgy_attr_lookup_by_id()

The **sec_rgy_attr_lookup_by_id()** call, which reads attributes for a specified object, uses a cursor of type **sec_attr_cursor_t**. This cursor must be allocated before it can be used as input to the **sec_rgy_attr_lookup_by_id()** call. In addition, it can also be initialized to the first attribute in the specified object's list of attributes, although this is not required. After use, the resources allocated to the **sec_attr_cursor_t** must be released.

The following calls allocate, initialize, and release a **sec_attr_cursor_t** for use with the **sec_rgy_attr_lookup_by_id()** call:

- **sec_rgy_attr_cursor_init()**

  The **sec_rgy_attr_sch_cursor_init()** call allocates resources to and initializes the cursor to the first attribute in the specified object's list of attributes. This call also supplies the total number of attributes attached to the object as part of its output. The cursor allocation is a local operation. The cursor initialization is a remote operation and makes a remote call to the registry.
- **sec_rgy_attr_cursor_alloc()**

  The **sec_rgy_attr_cursor_alloc()** call allocates resources to the cursor but does not initialize the cursor. However, since the **sec_rgy_attr_lookup_by_id()** call will initialize the cursor if it is passed in uninitialized, you may prefer this call to limit the number of remote calls performed by the application. Be aware that the **sec_rgy_attr_cursor_init()** call provides the total number of attributes attached to the specified object, a piece of information not provided by this call.
- **sec_rgy_attr_cursor_release()**

  The **sec_rgy_attr_cursor_release()** call releases all resources allocated to a **sec_attr_cursor_t** cursor used with the **sec_rgy_attr_lookup_by_id()** call.
- **sec_rgy_attr_cursor_reset()**

  The **sec_rgy_attr_cursor_reset()** call reinitializes a **sec_attr_cursor_t** cursor used with the **sec_rgy_attr_lookup_by_id()** call. The reset cursor can then be used without releasing and reallocating.

The sec_rgy_attr_lookup_by_id() Call

The **sec_rgy_attr_lookup_by_id()** call reads attributes specified by UUID for an object specified by name and domain. Specifically the call returns the following:

- An array of **sec_attr_t** values.
- A count of the total number of attribute instances returned.
- A count of the total number of attribute instances that could not be returned because of size constraints of the **sec_attr_t** array. (Note that the call allows the size of the array to be specified.)

For multivalued attributes, the call returns a **sec_attr_t** for each value as an individual attribute instance. For attribute sets, the call returns a **sec_attr_t** for each member of the set, but not the set instance. This routine is useful for programmatic access.

If the attribute instance to be read is not associated with a query trigger or no additional information is required by the query trigger, an attribute UUID is all that is required. For these attribute instances, supply the attribute UUID in the input array and set the attribute encoding (in **sec_attr_encoding_t**) to **sec_attr_enc_void**.

If the attribute instance to be read is associated with a query attribute trigger that requires additional information before it can process the query request, use a **sec_attr_value_t** to supply the requested information by doing the following:

- Set the **sec_attr_encoding_t** to an encoding type that is compatible with the information required by the query attribute trigger.
- Set the **sec_attr_value_t** to hold the required information.

You can define the number of elements in the input array of **sec_attr_t** values (in the *num_attr_keys* parameter). If you define the number of elements as 0 (zero), the call returns all of the object's attribute instances that the caller is authorized to see. You should be aware, however, that if you define the number of elements as zero and the attribute is associated with a query attribute trigger, you will be unable to pass any information to the query attribute trigger.

The sec_rgy_attr_set_lookup_by_id() Call

The **sec_rgy_attr_set_lookup_by_id()** call reads attribute sets specified by set instance UUID for an object specified by name and domain. Specifically the call returns the following:

- A **sec_attr_t** for each attribute instance in the attribute set.
- A count of the total number of attribute set instances returned.
- A count of the total number that could not be returned because of size constraints of the **sec_attr_t** array. (Note that the call allows the size and length of the array to be specified.)

Note: Since attribute triggers cannot be associated with an attribute set instance, this call provides no way to supply input data to a query attribute trigger.

The sec_rgy_attr_lookup_by_name() Call

The **sec_rgy_attr_lookup_by_name()** call reads a single attribute instance specified by name for an object specified by name and domain. The call returns a **sec_attr_t** for the specified attribute instance.

For multivalued attributes, the call returns the first instance of the multivalued attribute. (To retrieve every instance of a multivalued attribute, use the **sec_rgy_attr_lookup_by_id()** call.)

For attribute sets, the call returns the attribute set instance, not the member instances. To retrieve all members of the set, use the **sec_rgy_attr_lookup_by_id()** call.

Note: This call provides no way to supply input data to a query attribute trigger. If the attribute to be read is associated with a query trigger that requires input data, use the **sec_rgy_attr_lookup_by_id()** call.

The Attribute Trigger Facility

Some attribute types require the support of an outside server either to verify input attribute values or to supply output attribute values when those values are stored in an external database. Such a server could, for example, connect a legacy registry system to the DCE registry. The attribute trigger facility provides for automatic calls to outside DCE servers, known as attribute triggers.

Trigger servers, which are written by application developers, export the **sec_attr_trig** interface. They are invoked automatically when an attribute that has been associated with an attribute trigger (during schema entry creation) is queried or updated. The attribute trigger facility consists of three components:

- The attribute schema trigger fields (**trig_types** and **trig_binding**) that associate an attribute trigger and its binding information with an attribute type. These fields are part of the standard creation of a schema entry that defines an attribute type. See “Attribute Schema” on page 552.
- The **sec_attr_trig** APIs that define the query and update trigger operations. The APIs are provided in the **sec_attr_trig_ *()** calls.

- The user-written attribute trigger servers are independent from DCE servers. The trigger servers implement the trigger operations for the attribute types that require attribute trigger processing. These servers are not provided as part of DCE, but must be written by application developers.

Defining an Attribute Trigger/Attribute Association

When an attribute is created with the **sec_rgy_attr_update()** call, you define the association between the attribute type and an attribute trigger by specifying the following:

- Trigger Type

  Defines the trigger as a query server (invoked for query operations) or an update server (invoked for updates operations). The trigger type is defined in a **sec_attr_trig_type_t** data type, which is used by a **sec_attr_schema_entry_t** data type.
- Trigger Binding

  Defines the server binding handle for the attribute trigger. The details of the trigger binding are defined in a number of data types, which are also used by the **sec_attr_schema_entry_t** data type. Trigger binding is described in detail in “Trigger Binding” on page 557.

Only if both of pieces of information are provided will the association between the attribute type and the attribute trigger be created. You can associate an attribute trigger to any attribute type of any encoding except for attribute sets.

Query Triggers

When you execute a call that accesses an attribute associated with a query trigger, the client-side attribute lookup code performs the following tasks:

- Binds to the attribute trigger (using a binding from the attribute type’s schema entry)
- Makes the remote **sec_attr_trig_query()** call to the attribute trigger server, passing in the attribute keys and optional information provided by the caller
- If the **sec_attr_trig_query()** call is successful, returns the output attribute(s) to the caller

If you execute a **sec_rgy_attr()** update call with an attribute type that is associated with a query trigger, not an update trigger, the input attribute values are ignored and a “stub” attribute instance is created on the named object simply to mark the existence of this attribute on the object. Modifications to the real attribute value must occur at the attribute trigger.

Update Triggers

When you execute a call that accesses an attribute associated with an update trigger, the client-side attribute update code performs the following tasks:

- Binds to the attribute trigger (using a binding from the attribute type’s schema entry)
- Makes the remote **sec_attr_trig_update()** call to the attribute trigger server, passing in the attributes provided by the caller
- If the **sec_attr_trig_update()** call is successful, stores the output attribute(s) in the registry database and returns the output attribute(s) to the caller

Trigger Binding

Two data types are used to defined an attribute trigger. The **sec_attr_trig_type_t** type defines the type of attribute trigger. The **sec_attr_bind_info_t** data type, illustrated in Figure 79 and described in this section, specifies the attribute trigger's binding.

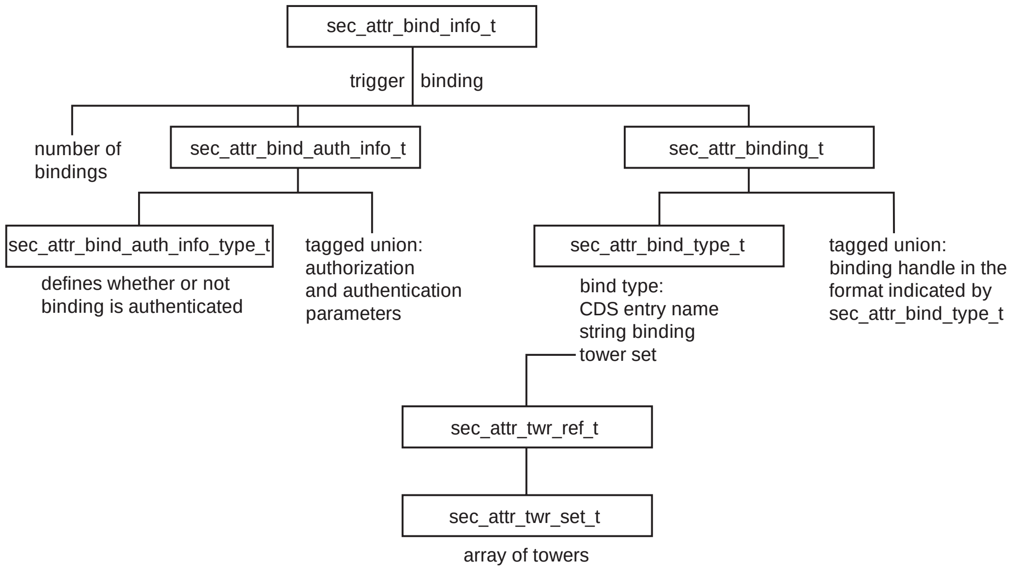

sec_attr_bind_info_t
trigger
binding
number of bindings
sec_attr_bind_auth_info_t
sec_attr_binding_t
sec_attr_bind_auth_info_type_t
defines whether or not binding is authenticated
tagged union: authorization and authentication parameters
sec_attr_bind_type_t
bind type: CDS entry name string binding tower set
tagged union: binding handle in the format indicated by sec_attr_bind_type_t
sec_attr_twr_ref_t
sec_attr_twr_set_t
array of towers

Figure 79. The sec_attr_bind_info_t Data Type

The **sec_attr_bind_info_t** data type uses two data types: **sec_attr_binding_t**, which defines the information used to generate binding handle and **sec_attr_bind_auth_info_t**, which defines the binding authentication and authorization information.

The sec_attr_binding_t Data Type

To describe the binding handle, the **sec_attr_binding_t** type uses a **sec_attr_bind_type_t** data type that specifies the format to the data used to generate the binding handle and a tagged union that contains the binding handle. The binding handle can be generated from any of the following:

- **A server directory entry name** (used with **rpc_ns_binding_import_ *()** calls)

  If the binding information is a server name, call **rpc_ns_binding_import_begin()** to establish a context for importing RPC binding handles from the name service database. For the **rpc_ns_binding_import_begin()** call, specify the CDS server directory entry name, an entry name syntax value of **rpc_c_ns_syntax_dce**, and **sec_attr_trig** as the interface handle of the interface to import.
- **A string binding** (used with **rpc_binding_from_string_binding()** calls)

  If the binding information is a string binding, call **rpc_binding_from_string_binding()** to generate an RPC binding handle.
- **An RPC protocol tower set** (used with **rpc_tower_to_binding()** calls)

  If the binding information is a protocol tower, two additional data types are used to pass in an unallocated array of towers, which the server must then allocate. These data types are **sec_attr_twr_ref_t** to point to the tower and **sec_attr_twr_set_t** to define the array of towers.

Architectural components of DCE can take advantage of the internal **rpc_tower_to_binding** operation in **rpcpvt.idl** to generate a binding handle from the canonical representation of a protocol tower.

Although the server directory entry name, with the actual server address stored in CDS, is the recommended way to specify an attribute trigger binding handle, prototype applications may want to specify a string binding or protocol tower for convenience.

The sec_attr_bind_auth_info_t Data Type

To describe whether or not RPC calls to the server will be authenticated and, for authenticated calls, to provide authentication and authorization information, the **sec_attr_bind_auth_info_t** type uses the **sec_attr_bind_auth_info_type_t** data type, and a tagged union. The **sec_attr_bind_auth_info_type_t** type defines whether or not the call is authenticated. The tagged union contains the authentication and authorization parameters.

Once a binding handle is obtained, call **rpc_binding_set_auth_info()** and supply it with the binding handle and authorization and authentication information.

Access Control on Attributes with Triggers

When a query or update call accesses an attribute associated with an attribute trigger, the call checks the ACL of the object with which the attribute is associated to see if the client has the permissions required for the operation. If access is granted, the operation returns a binding handle authenticated with the client’s login context. This handle is then used to perform the **sec_attr_trig_query** or **sec_attr_trig_update** operation.

Access to information maintained by an attribute trigger is controlled entirely by that attribute trigger. The attribute trigger can choose to implement any authorization mechanism, including none. For example, the attribute trigger can obtain the client’s identity from the RPC runtime to perform name-based authentication and perform ACL checks (or any other type of access control mechanism), and it can query the registry attribute schema for the attribute type’s permission set to use for an ACL check. Access control on attribute information stored outside of the registry database is left to the application designer.

Calls that Access Attribute Triggers

This section describes the calls that send information to and receive it from attribute triggers.

Using sec_attr_trig_cursor_t with sec_attr_trig_query()

The **sec_attr_trig_query()** call, which reads attributes associated with a query attribute trigger, uses a cursor of type **sec_attr_trig_cursor_t**. This cursor must be allocated and initialized before it can be used as input to the **sec_attr_trig_query()** call. After use, the resources allocated to **sec_attr_trig_cursor_t** must be released.

The following calls allocate, initialize, and release a **sec_attr_trig_cursor_t** type for use with the **sec_attr_trig_query()** call:

- **sec_attr_trig_cursor_init()**

The **sec_attr_trig_cursor_init()** call allocates resources to the cursor and initializes the cursor to the first attribute in the list of attributes for the object whose binding handle is specified. This call makes a remote call.

- **sec_attr_trig_cursor_release()**

  The **sec_rgy_attr_cursor_release()** call releases all resources allocated to a **sec_attr_trig_cursor_t** type by **sec_attr_trig_cursor_init()**.

The sec_rgy_attr_trig_query() and sec_rgy_attr_trig_update() Calls

The **sec_attr_trig_query()** call reads instances of attributes coded with a trigger type of query for a specified object. It passes an array of **sec_attr_t** values to a query attribute trigger and receives the output parameters back from the server. The **sec_attr_trig_update()** routine passes attributes coded with a trigger type of update to an update attribute trigger for evaluation before the updates are made to the registry.

Both calls are called automatically by the DCE attribute lookup or update code for all schema entries that specify a trigger. Although you should not call these calls directly, if you are implementing a trigger server, it will receive input from these calls and the attribute trigger's output should be passed back to them. The data received must be in a form accessible to the call and, if it is the result of an update, a form that can be stored in the registry database.

The object whose attribute instances are to be read or updated is identified by

- The name of the cell in which the object exists
- The name of the object or a UUID in string format that identifies the object

The priv_attr_triq_query() Call

The **priv_attr_trig_query()** call is used by the privilege service to retrieve trigger attributes and add them to a princpal's EPAC. The privilege service executes this call when it receives a request to add a principal and its extended attribute instances to an EPAC and the attributes are associated with a trigger server. The call passes an array of **sec_attr_t** values to the attribute trigger and receives the attribute values back from the trigger server in another array of **sec_attr_t** values. If the principal is being added to a delegation chain, the call also passes the UUIDs of all of the current members of the delegation chain to the trigger server. The trigger server can then evaluate all identities to determine access rights to the requested attributes.

Like the **sec_rgy_attr_trig_update()** calls, you will not call **priv_attr_trig_query()** directly. However, if you are implementing a trigger server, it will receive input from these calls and the attribute trigger's output should be passed back to the call. The data received must be in a form accessible to the call.

The DCE Attribute API

The DCE attribute calls are not described in detail. This is because, with the exception of the calls that bind to a selected database (**dce_attr_sch_bind()**(and **dce_attr_sch_bind_free()**), the **dce_sec_attr_ *()** calls are the same as the **sec_rgy_attr_sch_ *()** calls. Refer to “The ERA API” on page 551 for information on using each call. Note also that the DCE attribute calls are suffixed with **3dce**, not **3sec** (for example, **dce_attr_sch_bind.3dce**).

The DCE attribute API consists of the following calls:

- **dce_attr_sch_bind()**

  Returns an opaque handle of type **dce_attr_sch_handle_t** to a schema object specified by name and sets authentication and authorization parameters for the handle. This is the call used to bind to the schema of your choice.
- **dce_attr_sch_bind_free()**

  Releases an opaque handle of type **dce_attr_sch_handle_t**.
- **dce_attr_sch_create_entry()**

  Creates a schema entry in a schema bound to with **dce_attr_sch_bind**. This call is based on **sec_rgy_attr_sch_create_entry()** and is used in the same way.
- **dce_attr_sch_update_entry()**

  Updates a schema entry in a schema bound to with **dce_attr_sch_bind()**. This call is based on **sec_rgy_attr_sch_update_entry()** and is used in the same way.
- **dce_attr_sch_delete_entry()**

  Deletes a schema entry in a schema bound to with **dce_attr_sch_bind().** This call is based on **sec_rgy_attr_sch_delete_entry()** and is used in the same way.
- **dce_attr_sch_scan()**

  Reads a specified number of schema entries. This call is based on **sec_rgy_attr_sch_scan()** and is used in the same way.
- **dce_attr_sch_cursor_init()**

  Allocates resources to and initializes a cursor used with **dce_attr_sch_scan()**. The **dce_attr_sch_cursor_init()** routine makes a remote call that also returns the current number of schema entries in the schema. The **dce_attr_sch_cursor_init()** call is based on **sec_rgy_attr_sch_cursor_init()** and is used in the same way.
- **dce_attr_sch_cursor_alloc()**

  Allocates resources to a cursor used with **dce_attr_sch_scan()**. The **dce_attr_sch_cursor_alloc()** routine is a local operation. The **dce_attr_sch_cursor_alloc()** call is based on **sec_rgy_attr_sch_cursor_alloc()** and is used in the same way.
- **dce_attr_sch_cursor_release()**

  Releases states associated with a cursor created by **dce_attr_sch_cursor_alloc()** or **dce_attr_sch_cursor_init()**. The **dce_attr_sch_cursor_release()** call is based on **sec_rgy_attr_sch_cursor_release()** and is used in the same way.
- **dce_attr_sch_cursor_reset()**

  Reinitializes a cursor used with **dce_attr_sch_scan()**. The reset cursor can then be reused without releasing and reallocating. This call is based on the **sec_rgy_attr_sch_cursor_reset()** and is used in the same way.
- **dce_attr_sch_lookup_by_id()**

  Reads a schema entry identified by UUID. This call is based on **sec_rgy_attr_lookup_by_id()** and is used in the same way.
- **dce_attr_sch_lookup_by_name()**

  Reads a schema entry identified by name. This call is based on **sec_rgy_attr_sch_lookup_by_name()** and is used in the same way.
- **dce_attr_sch_get_acl_mgrs()**

  Retrieves the UUIDs of ACL manager types protecting objects dominated by a named schema. This call is based on **sec_rgy_attr_sch_get_acl_mgrs()** and is used in the same way.

- **dce_attr_sch_aclmgr_strings()**

  Retrieves the print strings containing information about ACL manager types protecting objects dominated by a named schema. The print strings contain the manager's name, help information, and supported permission bits. This call is based on **sec_rgy_attr_sch_aclmgr_strings()** and is used in the same way.

Macros to Aid Extended Attribute Programming

The extended attribute APIs includes macros to help programmers using the extended attribute interfaces. The macros perform a variety of functions including

- Accessing fields in data structures
- Calculating the size of data structures
- Performing semantic and flag checks
- Setting flags

The macros are in **dce/rpcbase.h**, which is derived from **dce/rpcbase.idl**.

The following subsections list the definitions of each macro.

Macros to Access Binding Fields

In the following macro definitions, which are used by a **sec_attr_schema_entry_t** and its equivalent **dce_attr_sch** data type, B is a pointer to **sec_attr_bind_info_t**.

```
#define SA_BND_AUTH_INFO(B) (B)->auth_info
#define SA_BND_AUTH_INFO_TYPE(B) (SA_BND_AUTH_INFO(B)).info_type

#define SA_BND_AUTH_SVR_PNAME_P(B) \

(SA_BND_AUTH_DCE_INFO(B)).svr_princ_name

#define SA_BND_AUTH_PROT_LEVEL(B) \

    (SA_BND_AUTH_DCE_INFO(B)).protect_level

#define SA_BND_AUTH_AUTHN_SVC(B) \

     (SA_BND_AUTH_DCE_INFO(B)).authn_svc

#define SA_BND_AUTH_AUTHZ_SVC(B) \

(SA_BND_AUTH_DCE_INFO(B)).authz_svc

#define SA_BND_NUM(B) (B)->num_bindings
#define SA_BND_ARRAY(B,I) (B)->bindings[I]
#define SA_BND_TYPE(B,I) (SA_BND_ARRAY(B,I)).bind_type

#define SA_BND_STRING_P(B,I) \

(SA_BND_ARRAY(B,I)).tagged_union.string_binding

#define SA_BND_SVRNAME_P(B,I) \

(SA_BND_ARRAY(B,I)).tagged_union.svrname

#define SA_BND_SVRNAME_SYNTAX(B,I) \

(SA_BND_SVRNAME_P(B,I))->name_syntax

#define SA_BND_SVRNAME_NAME_P(B,I) \
(SA_BND_SVRNAME_P(B,I))->name

#define SA_BND_TWRSET_P(B,I) \

(SA_BND_ARRAY(B,I)).tagged_union.twr_set

#define SA_BND_TWRSET_COUNT(B,I) (SA_BND_TWRSET_P(B,I))->count
#define SA_BND_TWR_P(B,I,J) (SA_BND_TWRSET_P(B,I))->towers[J]
#define SA_BND_TWR_LEN(B,I,J) (SA_BND_TWR_P(B,I,J))->tower_length

#define SA_BND_TWR_OCTETS(B,I,J) \


(SA_BND_TWR_P(B,I,J))->tower_octet_string
```

Macros to Access Schema Entry Fields

In the following macro definitions, S is a pointer to **sec_attr_schema_entry_t** (and its equivalent **dce_attr_sch** data type) and I and J are nonnegative integers for array element selection.

```
#define SA_ACL_MGR_SET_P(S)       (S)->acl_mgr_set
#define SA_ACL_MGR_NUM(S)        (SA_ACL_MGR_SET_P(S))->num_acl_mgrs
#define SA_ACL_MGR_INFO_P(S,I)     (SA_ACL_MGR_SET_P(S))->mgr_info[I]
#define SA_ACL_MGR_TYPE(S,I)      (SA_ACL_MGR_INFO_P(S,I))->acl_mgr_type
#define SA_ACL_MGR_PERMS_QUERY(S,I)   (SA_ACL_MGR_INFO_P(S,I))->query_permset
#define SA_ACL_MGR_PERMS_UPDATE(S,I)  (SA_ACL_MGR_INFO_P(S,I))->update_permset
#define SA_ACL_MGR_PERMS_TEST(S,I)   (SA_ACL_MGR_INFO_P(S,I))->test_permset
#define SA_ACL_MGR_PERMS_DELETE(S,I)  (SA_ACL_MGR_INFO_P(S,I))->delete_permset
#define SA_TRG_BND_INFO_P(S)      (S)->trig_binding

#define SA_TRG_BND_AUTH_INFO(S) \

(SA_BND_AUTH_INFO(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_INFO_TYPE(S) \

(SA_BND_AUTH_INFO_TYPE(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_DCE_INFO(S) \

(SA_BND_AUTH_DCE_INFO(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_SVR_PNAME_P(S) \

(SA_BND_AUTH_SVR_PNAME_P(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_PROT_LEVEL(S) \

(SA_BND_AUTH_PROT_LEVEL(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_AUTHN_SVC(S) \

(SA_BND_AUTH_AUTHN_SVC(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_AUTHZ_SVC(S) \

(SA_BND_AUTH_AUTHZ_SVC(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_NUM(S) \

(SA_BND_NUM(SA_TRG_BND_INFO_P(S)))
#define SA_TRG_BND_ARRAY(S,I) \

(SA_BND_ARRAY((SA_TRG_BND_INFO_P(S)),I))
```

```
#define SA_TRG_BND_TYPE(S,I) \

(SA_BND_TYPE((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_STRING_P(S,I) \

(SA_BND_STRING_P((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_SVRNAME_P(S,I) \

(SA_BND_SVRNAME_P((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_SVRNAME_SYNTAX(S,I) \

    (SA_BND_SVRNAME_SYNTAX((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_SVRNAME_NAME_P(S,I) \

    (SA_BND_SVRNAME_NAME_P((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_TWRSET_P(S,I) \

(SA_BND_TWRSET_P((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_TWRSET_COUNT(S,I) \

(SA_BND_TWRSET_COUNT((SA_TRG_BND_INFO_P(S)),I))

#define SA_TRG_BND_TWR_P(S,I,J) \

(SA_BND_TWR_P((SA_TRG_BND_INFO_P(S)),I,J))

#define SA_TRG_BND_TWR_LEN(S,I,J) \

(SA_BND_TWR_LEN((SA_TRG_BND_INFO_P(S)),I,J))

#define SA_TRG_BND_TWR_OCTETS(S,I,J) \


(SA_BND_TWR_OCTETS((SA_TRG_BND_INFO_P(S)),I,J))
```

Macros to Access Attribute Instance Fields

In the following macro descriptions, S is a pointer to **sec_attr_t**, and I and J are nonnegative integers for array element selection.

```
#define SA_ATTR_ID(S) (S)->attr_id
#define SA_ATTR_VALUE(S) (S)->attr_value
#define SA_ATTR_ENCODING(S) (SA_ATTR_VALUE(S)).attr_encoding

#define SA_ATTR_INTEGER(S) \

(SA_ATTR_VALUE(S)).tagged_union.signed_int

#define SA_ATTR_PRINTSTRING_P(S) \

(SA_ATTR_VALUE(S)).tagged_union.printstring


#define SA_ATTR_STR_ARRAY_P(S) \

(SA_ATTR_VALUE(S)).tagged_union.string_array

#define SA_ATTR_STR_ARRAY_NUM(S) (SA_ATTR_STR_ARRAY_P(S))->num_strings
```

```
#define SA_ATTR_STR_ARRAY_ELT_P(S,I) (SA_ATTR_STR_ARRAY_P(S))->strings[I]

#define SA_ATTR_BYTES_P(S) \

(SA_ATTR_VALUE(S)).tagged_union.bytes

#define SA_ATTR_BYTES_LEN(S) (SA_ATTR_BYTES_P(S))->length
#define SA_ATTR_BYTES_DATA(S,I) (SA_ATTR_BYTES_P(S))->data[I]

#define SA_ATTR_IDATA_P(S) \

(SA_ATTR_VALUE(S)).tagged_union.idata

#define SA_ATTR_IDATA_CODESET(S) (SA_ATTR_IDATA_P(S))->codeset
#define SA_ATTR_IDATA_LEN(S) (SA_ATTR_IDATA_P(S))->length
#define SA_ATTR_IDATA_DATA(S,I) (SA_ATTR_IDATA_P(S))->data[I]

#define SA_ATTR_UUID(S) \

(SA_ATTR_VALUE(S)).tagged_union.uuid

#define SA_ATTR_SET_P(S) \

(SA_ATTR_VALUE(S)).tagged_union.attr_set
#define SA_ATTR_SET_NUM(S) (SA_ATTR_SET_P(S))->num_members
#define SA_ATTR_SET_MEMBERS(S,I) (SA_ATTR_SET_P(S))->members[I]

#define SA_ATTR_BND_INFO_P(S) \

(SA_ATTR_VALUE(S)).tagged_union.binding

#define SA_ATTR_BND_AUTH_INFO(S) \

(SA_BND_AUTH_INFO(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_INFO_TYPE(S) \

(SA_BND_AUTH_INFO_TYPE(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_DCE_INFO(S) \

(SA_BND_AUTH_DCE_INFO(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_SVR_PNAME_P(S) \

(SA_BND_AUTH_SVR_PNAME_P(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_PROT_LEVEL(S) \

(SA_BND_AUTH_PROT_LEVEL(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_AUTHN_SVC(S) \

(SA_BND_AUTH_AUTHN_SVC(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_AUTHZ_SVC(S) \

(SA_BND_AUTH_AUTHZ_SVC(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_NUM(S) \

(SA_BND_NUM(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_ARRAY(S,I) \

(SA_BND_ARRAY((SA_ATTR_BND_INFO_P(S)),I))
```

```
#define SA_ATTR_BND_TYPE(S,I) \

(SA_BND_TYPE((SA_ATTR_BND_INFO_P(S)),I))
#define SA_ATTR_BND_STRING_P(S,I) \

(SA_BND_STRING_P((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_SVRNAME_P(S,I) \

(SA_BND_SVRNAME_P((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_SVRNAME_SYNTAX(S,I) \

    (SA_BND_SVRNAME_SYNTAX((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_SVRNAME_NAME_P(S,I) \

    (SA_BND_SVRNAME_NAME_P((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_TWRSET_P(S,I) \

(SA_BND_TWRSET_P((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_TWRSET_COUNT(S,I) \

(SA_BND_TWRSET_COUNT((SA_ATTR_BND_INFO_P(S)),I))

#define SA_ATTR_BND_TWR_P(S,I,J) \

(SA_BND_TWR_P((SA_ATTR_BND_INFO_P(S)),I,J))

#define SA_ATTR_BND_TWR_LEN(S,I,J) \

(SA_BND_TWR_LEN((SA_ATTR_BND_INFO_P(S)),I,J))

#define SA_ATTR_BND_TWR_OCTETS(S,I,J) \


(SA_BND_TWR_OCTETS((SA_ATTR_BND_INFO_P(S)),I,J))
```

Binding Data Structure Size Calculation Macros

The following macros are supplied to calculate the size of data types that hold binding information. The macros work with the ERA API data types and their equivalent **dce_attr_sch** data types.

```
/*
 * SA_BND_INFO_SIZE(N) - calculate the size required
 * for a sec_attr_bind_info_t with N bindings.
 */
#define SA_BND_INFO_SIZE(N) (sizeof(sec_attr_bind_info_t) + \

    (((N) - 1) * sizeof(sec_attr_binding_t)))
/*
 * SA_TWR_SET_SIZE(N) - calculate the size required
 * for a sec_attr_twr_set_t with N towers.
 */
#define SA_TWR_SET_SIZE(N) (sizeof(sec_attr_twr_set_t) + \

    (((N) - 1) * sizeof(sec_attr_twr_ref_t)))

/*
 * SA_TWR_SIZE(N) - calculate the size required
 * for a twr_t with a tower_octet_string of length N.
 */
#define SA_TWR_SIZE(N) (sizeof(twr_t) + (N) - 1)
```

Schema Entry Data Structure Size Calculation Macros

The following macro is supplied to calculate the size of a **sec_attr_alc_mgr_info_set_t** data type.

```
/*
 * SA_ACL_MGR_SET_SIZE(N) - calculate the size required
 * for a sec_attr_acl_mgr_info_set_t with N acl_mgrs.
 */
#define SA_ACL_MGR_SET_SIZE(N) (sizeof(sec_attr_acl_mgr_info_set_t) + \

    (((N) - 1) * sizeof(sec_attr_acl_mgr_info_p_t))
)
```

Attribute Instance Data Structure Size Calculation Macros

The following macros are supplied to calculate the size of data types that hold attribute information.

```
/*
 * SA_ATTR_STR_ARRAY_SIZE(N) - calculate the size required
 * for a sec_attr_enc_str_array_t with N sec_attr_enc_printstring_p_t-s.
 */
#define SA_ATTR_STR_ARRAY_SIZE(N) (sizeof(sec_attr_enc_str_array_t) + \

    (((N) - 1) * sizeof(sec_attr_enc_printstring_p_t)))
/*
 * SA_ATTR_BYTES_SIZE(N) - calculate the size required
 * for a sec_attr_enc_bytes_t with byte string length of N.
 */
#define SA_ATTR_BYTES_SIZE(N) (sizeof(sec_attr_enc_bytes_t) + (N) - 1)

/*
 * SA_ATTR_IDATA_SIZE(N) - calculate the size required
 * for a sec_attr_i18n_data_t with byte string length of N.
 */
#define SA_ATTR_IDATA_SIZE(N) (sizeof(sec_attr_i18n_data_t) + (N) - 1)

/*
 * SA_ATTR_SET_SIZE(N) - calculate the size required
 * for a sec_attr_enc_attr_set_t with N members (uuids).
 */
#define SA_ATTR_SET_SIZE(N) (sizeof(sec_attr_enc_attr_set_t) + \

    (((N) - 1) * sizeof(uuid_t))
)
```

Binding Semantic Check Macros

The following macros are supplied to check the semantics of entries in the binding fields. The macros work with the ERA API data types and their equivalent **dce_attr_sch** data types.

```
/*
 * SA_BND_AUTH_INFO_TYPE_VALID(B) - evaluates to TRUE (1)
 * if the binding auth_info type is valid; FALSE (0) otherwise.
 * B is a pointer to a sec_attr_bind_info_t.
 */
```

```
#define SA_BND_AUTH_INFO_TYPE_VALID(B) ( \

  (SA_BND_AUTH_INFO_TYPE(B)) == sec_attr_bind_auth_none || \

  (SA_BND_AUTH_INFO_TYPE(B)) == sec_attr_bind_auth_dce ? true : false
)

/*
 * SA_BND_AUTH_PROT_LEV_VALID(B) - evaluates to TRUE (1)
 * if the binding auth_info protect_level is valid; FALSE (0) otherwise.
 * B is a pointer to a sec_attr_bind_info_t.
 */
#define SA_BND_AUTH_PROT_LEV_VALID(B) ( \

  (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_default || \

  (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_none || \

  (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_connect || \

  (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_call || \

  (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_pkt || \

  (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_pkt_integ || \

  (SA_BND_AUTH_PROT_LEVEL(B)) == rpc_c_protect_level_pkt_privacy ? \

  true : false)

/*
 * SA_BND_AUTH_AUTHN_SVC_VALID(B) - evaluates to TRUE (1)
 * if the binding auth_info authentication service is valid;
 * FALSE (0) otherwise.
 * B is a pointer to a sec_attr_bind_info_t.
 */
#define SA_BND_AUTH_AUTHN_SVC_VALID(B) ( \

  (SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_none || \

  (SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_dce_secret || \

  (SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_dce_public || \

  (SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_dce_dummy || \

  (SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_dssa_public || \

  (SA_BND_AUTH_AUTHN_SVC(B)) == rpc_c_authn_default ? \
  true : false)

/*
 * SA_BND_AUTH_AUTHZ_SVC_VALID(B) - evaluates to TRUE (1)
 * if the binding auth_info authorization service is valid;
 * FALSE (0) otherwise.
 * B is a pointer to a sec_attr_bind_info_t.
 */
#define SA_BND_AUTH_AUTHZ_SVC_VALID(B) ( \

  (SA_BND_AUTH_AUTHZ_SVC(B)) == rpc_c_authz_none || \

  (SA_BND_AUTH_AUTHZ_SVC(B)) == rpc_c_authz_name || \

  (SA_BND_AUTH_AUTHZ_SVC(B)) == rpc_c_authz_dce ? \

  true : false)
```

Schema Entry Semantic Check Macros

The following macros are supplied to check the semantics of schema entry fields. In the macros, S is a pointer to **sec_attr_schema_entry_t** and its equivalent **dce_attr_sch** data type.

```
#define SA_TRG_BND_AUTH_INFO_TYPE_VALID(S) \

(SA_BND_AUTH_INFO_TYPE_VALID(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_PROT_LEV_VALID(S) \

(SA_BND_AUTH_PROT_LEV_VALID(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_AUTHN_SVC_VALID(S) \

(SA_BND_AUTH_AUTHN_SVC_VALID(SA_TRG_BND_INFO_P(S)))

#define SA_TRG_BND_AUTH_AUTHZ_SVC_VALID(S) \

(SA_BND_AUTH_AUTHZ_SVC_VALID(SA_TRG_BND_INFO_P(S))
```

Attribute Instance Semantic Check Macros

The following macros are supplied to check the semantics of entries in the attribute instance fields. In the following macros, S is a pointer to **sec_attr_t**. F is a **sec_attr_trigs_types_flags_t**.

```
#define SA_ATTR_BND_AUTH_INFO_TYPE_VALID(S) \

(SA_BND_AUTH_INFO_TYPE_VALID(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_PROT_LEV_VALID(S) \

(SA_BND_AUTH_PROT_LEV_VALID(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_AUTHN_SVC_VALID(S) \

(SA_BND_AUTH_AUTHN_SVC_VALID(SA_ATTR_BND_INFO_P(S)))

#define SA_ATTR_BND_AUTH_AUTHZ_SVC_VALID(S) \

(SA_BND_AUTH_AUTHZ_SVC_VALID(SA_ATTR_BND_INFO_P(S))

#define SA_SCH_FLAG_IS_SET(S,F) \

(((S)->schema_entry_flags & (F)) == (F))

#define SA_SCH_FLAG_IS_SET_UNIQUE(S) \

(SA_SCH_FLAG_IS_SET((S),sec_attr_sch_entry_unique))

#define SA_SCH_FLAG_IS_SET_MULTI_INST(S) \

(SA_SCH_FLAG_IS_SET((S),sec_attr_sch_entry_multi_inst))

#define SA_SCH_FLAG_IS_SET_RESERVED(S) \

(SA_SCH_FLAG_IS_SET((S),sec_attr_sch_entry_reserved))

#define SA_SCH_FLAG_IS_SET_USE_DEFAULTS(S) \


(SA_SCH_FLAG_IS_SET((S),sec_attr_sch_entry_use_defaults))
```

Schema Entry Flag Set and Unset Macros

The following macros set and unset flag(s) in the schema entry **schema_entry_flags** field. In the following macros, S is a pointer to **sec_attr_schema_entry_t**.

```
/*
 * Macros to set the flags.
 */
#define SA_SCH_FLAG_SET(S, FLAG) ((S)->schema_entry_flags |= (FLAG))

#define SA_SCH_FLAG_SET_UNIQUE(S) \

    (SA_SCH_FLAG_SET((S),sec_attr_sch_entry_unique))

#define SA_SCH_FLAG_SET_MULTI_INST(S) \

    (SA_SCH_FLAG_SET((S),sec_attr_sch_entry_multi_inst))

#define SA_SCH_FLAG_SET_RESERVED(S) \

    (SA_SCH_FLAG_SET((S),sec_attr_sch_entry_reserved))

#define SA_SCH_FLAG_SET_USE_DEFAULTS(S) \

    (SA_SCH_FLAG_SET((S),sec_attr_sch_entry_use_defaults))

/*
 * Macros to unset the flags.
 */
#define SA_SCH_FLAG_UNSET(S, FLAG) ((S)->schema_entry_flags \
    &= ~(FLAG))

#define SA_SCH_FLAG_UNSET_UNIQUE(S) \

    (SA_SCH_FLAG_UNSET((S),sec_attr_sch_entry_unique))

#define SA_SCH_FLAG_UNSET_MULTI_INST(S) \

    (SA_SCH_FLAG_UNSET((S),sec_attr_sch_entry_multi_inst))

#define SA_SCH_FLAG_UNSET_RESERVED(S) \

    (SA_SCH_FLAG_UNSET((S),sec_attr_sch_entry_reserved))

#define SA_SCH_FLAG_UNSET_USE_DEFAULTS(S) \

(SA_SCH_FLAG_UNSET((S),sec_attr_sch_entry_use_defaults))
```

Schema Trigger Entry Flag Check Macros

The following macros evaluate to TRUE if the requested flag(s) is set in the schema entry **trig_types** field. In the following macros, S is a pointer to **sec_attr_schema_entry_t** and F is a **sec_attr_trigs_types_flags_t** type.

```
#define SA_SCH_TRIG_FLAG_IS_SET(S,F) \

(((S)->trig_types & (F)) == (F))

#define SA_SCH_TRIG_FLAG_IS_NONE(S) \

(SA_SCH_TRIG_FLAG_IS_SET((S),sec_attr_trig_type_none))

#define SA_SCH_TRIG_FLAG_IS_QUERY(S) \
(SA_SCH_TRIG_FLAG_IS_SET((S),sec_attr_trig_type_query))

#define SA_SCH_TRIG_FLAG_IS_UPDATE(S) \

 (SA_SCH_FLAG_IS_SET((S),sec_attr_trig_type_update))
```

Utilities to Use with Extended Attribute Calls

The extended attribute APIs includes utilities to help programmers using the extended attribute interfaces. These utilities are

- **sec_attr_util_alloc_copy**—Copies data from one **sec_attr_t** data type to another.
- **sec_attr_util_free**—Frees memory allocated to **sec_attr_t** by the **sec_attr_util_alloc_copy()** function.
- **sec_attr_util_inst_free_ptrs**—Frees nonnull pointers in a **sec_attr_t** type.
- **sec_attr_util_inst_free**—Frees nonnull pointers in a **sec_attr_t** type and the pointer to the **sec_attr_t** itself.
- **sec_attr_util_sch_ent_free_ptrs**—Frees nonnull pointers in a **sec_attr_schema_entry_t type.**
- **sec_attr_util_sch_ent_free**—Frees nonnull pointers in a **sec_attr_schema_entry_t**type and the pointer to the **sec_attr_schema_entry_t** itself. The utility also works with the equivalent **dce_attr_sch** data type.

Chapter 30. The Login API

The login API communicates with the security server to establish, and possibly change, a principal’s login context. A login context contains the information necessary for a principal to qualify for (although not necessarily be granted) access to network services and possibly local resources as well. Login context information normally includes the following:

- Identity information concerning the principal, including its certificate of identity (in shared-secret authentication, this is the TGT), its PAC, and registry policy information such as the maximum lifetime of certificates of identity.
- The context state; that is, whether the authentication service has validated the context or not.
- The source of authentication information. (It may originate from the network authentication service, or locally, if that network service is unavailable.)

Establishing Login Contexts

This section outlines the basic procedure by which a network login context is established. See “Chapter 24. Authentication” on page 493 for a detailed description of this process.

The procedure is as follows:

1. The client calls **sec_login_setup_identity()** specifying the name of the principal whose network identity is to be established. Memory is allocated to receive the principal’s login context.
2.

   The client calls **sec_login_valid_and_cert_ident()**, which does the following:
   a. Forwards a TGT request encrypted with the user’s secret key and with a random key, to the authentication service, which decrypts the request, authenticates the principal, and returns a TGT for the principal.
   b. The client’s security runtime then decrypts the TGT and forwards it to the privilege service, which creates a PAC for the principal and encloses it in a PTGT, which is returned to the client’s security runtime.
   c. The runtime decrypts the message containing the PTGT and returns information about the source of the authentication information to the API. (If the authentication information comes from the network security server, then the login context is validated.)
3. Finally, the client invokes **sec_login_set_context()**, which enables child processes spawned from the calling process to inherit the validated context.

In the walkthrough of user authentication in “Chapter 24. Authentication” on page 493, we mentioned that one of the functions of **sec_login_valid_and_cert_ident()** is to demonstrate that a valid trust path exists between the authentication service and the host computer on which the principal is logging in. After setting up and validating a login context, any application that sets identity information for local processes should check to be sure that the server that provided the certificate of identity is legitimate in order to demonstrate that the trust path between the client and the authentication service is valid.

Validating the Login Context and Certifying the Security Server

Whereas a validated login context is one that is regarded as legitimate by the local security runtime, a validated and certified login context is one that is not only regarded as legitimate but also can be demonstrated to have been (in all likelihood, that is) issued by a legitimate security server. Certifying that the security server is legitimate prevents faked identity information from being propagated to local processes. For example, a spurious server could collaborate with a dishonest user in order to obtain an identity that conferred comprehensive permissions (for example, the **root** identity). With such an identity, the dishonest user could gain access to sensitive local objects, such as key-storage files for server principals that run on the host. (Servers running on other hosts would not trust this principal, however, because it does not know their keys.) Of course, if a spurious server can return to the application a ticket encrypted with the host's secret key, it means the server has access to the host's key; but, if this is the case, network security has already been seriously undermined.

When an application needs to certify the originator of a certificate of identity, it may call **sec_login_certify_identity()**. This routine makes an authenticated remote procedure call to the local security validation service of the **dced** daemon in order to acquire a ticket to the host principal. If **dced** succeeds in decrypting the message containing the ticket, then the server that granted the certificate of identity must know the host principal's secret key; this evidence indicates that it is a legitimate security server. Since **dced** runs with the identity **root** (in order to access the host's key), the process calling **sec_login_certify_identity()** need not.

The **sec_login_valid_and_cert_ident()** is similar to **sec_login_certify_identity()**, except that it combines the validation and certification procedures (and therefore, the password of the principal that is logging in must be known to the process making this call). The **sec_login_valid_and_cert_ident()** routine calls the security server for a ticket to the host and attempts decryption. The process calling **sec_login_valid_and_cert_ident()** must have access to the host's secret key, and so must run as **root**.

Note: Because system login programs should not set local identities derived from an uncertified context, all login API routines that return data from an uncertified context issue a warning.

Validating the Login Context Without Certifying the Security Server

An application that does not use login contexts to set local identity information does not need to certify its login contexts. Since an illegitimate security server is unlikely to know the key of a remote server principal with which the application may communicate, the application will simply be refused the service requested from the remote server principal. If local operating system identity information is assumed to be neither of interest nor of concern to an application, it may call **sec_login_validate_identity()**, which does not attempt to verify the security server's knowledge of the host principal's key.

The **sec_login_validate_identity()** routine does not acquire a PTGT, unlike the **sec_login_certify_identity()** and **sec_login_valid_and_cert_ident()** routines. Instead, the PTGT is acquired when the application first makes an authenticated remote procedure call.

Example of a System Login Program

Following is an example of a system login program that obtains a login context that can be trusted for both network and local operations.

Note: One of the function calls that appears in the following example, **sec_login_purge_context()**, is described in "Releasing and Purging a Context" on page 586.

```
if (sec_login_setup_identity(principal,sec_login_no_flags,
  &login_context,&st))
{
  ...get password...
  if (sec_login_valid_and_cert_ident(login_context, password,
    &reset_passwd, &auth_src,&st))
  {
   if(auth_src==sec_login_auth_src_network)
   {
     if (GOOD_STATUS(&st)
      sec_login_set_context(login_context);
   }
  }
  if (reset_passwd)
  {
   ...reset the user's password...

   if (passwd_reset_fails)
   {
     sec_login_purge_context(login_context)

     ...application login-failure actions...
   }

   ...application-specific login-valid actions...
  }
}
```

Context Inheritance

A process inherits the login context of its parent process unless the child process is associated with a principal that has logged in and so established a separate login context. The following subsections describe two additional aspects of context inheritance:

- How the initial context is established.
- How a process may inhibit context inheritance.

The Initial Context

An application invokes **sec_login_setup_identity()** so that it can then make other authenticated RPC calls. However, **sec_login_setup_identity()** is itself a local interface to an authenticated remote procedure call, and authenticated RPC needs a validated login context in order to execute. For applications like system login, the daemon **dced** supplies the validated context. However, a daemon that is started before **dced** is running on the host needs to be able to assume its host's identity. The initial context is established at boot time with **sec_login_init_first()**, which establishes the default context inheritance for processes running on the host. The routines **sec_login_setup_first()** and **sec_login_validate_first()** then set up and validate the context in a procedure like that used for user context validation.

Private Contexts

A process may inhibit context inheritance by setting a flag in **sec_login_setup_identity()**. If the flag indicates that the login context is private, then children of the calling process cannot inherit it. A child process can neither set a private context (since it is the function of **sec_login_set_context()** to make the context inheritable) nor export it to any other process.

Handling Expired Certificates of Identity

For a dishonest principal to make use of an intercepted certificate of identity, it must succeed in decrypting it. In order to make the task of decryption more difficult, a certificate of identity has a limited lifespan; and, once it expires, the associated login context is no longer valid.

Because this security feature may inconvenience users, an application may wish to warn a user when the certificate of identity is about to expire. The **sec_login_get_expiration()** routine returns the expiration date of a certificate of identity. When a certificate of identity is about to expire, the application may call **sec_login_refresh_identity()**, which may be used to refresh any login context.

Similarly, a server principal may need to determine whether a certificate of identity may expire during some long network operation and, if the certificate of identity is likely to expire, refresh it to ensure that the operation is not prevented from completion. Following is an example:

```
sec_login_get_expiration (login_context,&expire_time,&st);

if (expire_time < (current_time + operation_duration))
{
  if (sec_login_refresh_identity(login_context,&st))
  {
   ...identity has changed and must be validated again...
  }
  else
  {
   ...login context cannot be renewed...

   exit(0);
  }
}

operation();
```

Because **sec_login_refresh_identity()** acquires a certificate of identity, refreshed contexts must be revalidated with **sec_login_validate_identity()** or **sec_login_valid_and_cert_ident()** before they can be used.

The expiration date of a login context has no meaning with respect to local identity information; for the same reason, **sec_login_refresh_identity()** cannot refresh a login context that has been authenticated locally.

Importing and Exporting Contexts

Under some circumstances, an application may need two processes to run using the same login context. A process may acquire its login context in a form suitable for imparting to another process by calling **sec_login_export_context()**. This call collects the login context from the local context cache and loads it into a buffer. Another process may then call **sec_login_import_context()** to unpack the buffer and create its own login context cache to store the imported context. Since the context has already been validated, the process that imports it may use it immediately. (The CDS clerk is an example of a context importer.)

These operations are strictly local; that is, the exporting and importing processes must be running on the same host. In addition, a process cannot export a private context.

Changing a Groupset

The **sec_login_newgroups()** routine enables a principal to assume the minimum groupset that is required to accomplish a given task. For example, a user may have privilege attributes that include membership in an administrative group associated with a comprehensive permission set, and membership in a user group associated with a more restricted permission set. Such a user may not want the permissions associated with the administrative group, except when those permissions are essential to an administrative task (so as to avoid inadvertent damage to objects that are accessible to members of the administrative group, but not to members of the user group).

To offer users the capability of removing groups from their groupsets, an application may use the login API as shown in the following example.

Note: Two of the function calls that appear in the following example, **sec_login_get_current_context()** and **sec_login_inquire_net_info()**, are described in the following section.

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

sec_login_inquire_net_info(login_context,&net_info,&st);

for (i=0; i < num_groups; i++)
{
  ... query whether user wants to discard any current group
  memberships. Copy new group set to new_groups array ...
}

if (!sec_login_newgroups(login_context,sec_login_no_flags,
  num_new_groups, new_groups, &restricted_context,&st))
{
  if (st == sec_login_s_groupset_invalid)

   printf("Newgroupsetinvalid\n");

  ...application-specific error handling...
}
```

Note that the **sec_login_newgroups()** call can only return a restricted groupset: it cannot return a groupset larger than the one associated with the login context that is passed to it. This routine also enables the calling process to flag the new login context as private to the calling process.

Miscellaneous Login API Functions

The following subsections describe a few miscellaneous login API routines, some of which have appeared previously in examples in this chapter.

Getting the Current Context

The **sec_login_get_current_context()** routine returns a handle to the login context for the currently established principal. This routine is useful for several login API functions that take a login context handle as input.

Getting Information from a Login Context

The **sec_login_inquire_net_info()** routine returns a data structure comprising the principal's PAC, account expiration date, password expiration date, and identity expiration date. The **sec_login_free_net_info()** frees the memory allocated to this data structure.

Getting Password and Group Information for Local Process Identities

Two calls, **sec_login_get_pwent()** and **sec_login_get_groups()**, are useful for setting the local identity of a process. These routines return password or group information from the network registry, if that service is available, or from the local files of password and group information, if the network service is unavailable.

Releasing and Purging a Context

When a process is finished using a login context, it may call **sec_login_release_context()** to free storage occupied by the context handle. When a process releases a login context, the context is still available to other processes that use it. If an application needs to destroy a login context, it may call **sec_login_purge_context()**, which also frees storage occupied by the handle. Since a destroyed context is unavailable to all processes that use it, application developers should be careful when using **sec_login_purge_context()**.

Chapter 31. The Key Management API

Every principal has an entry in the registry database that specifies a secret key. In the case of an interactive principal (that is, a user), the secret key is derived from the principal's password. Just as users need to keep their passwords secure by memorizing them (rather than writing them down, for example), a noninteractive principal also needs to be able to store and retrieve its secret key in a secure manner. The key management API provides simple key management functions for noninteractive principals.

While the key management routines themselves are relatively secure, it is up to the application to ensure the security of the file or other device used to store the key. By default, server principals that run on the same computer share a local key file; however, the key management API also allows principals to specify an alternative local file.

When users change their passwords, they are free to forget their old passwords. When a noninteractive principal changes its secret key, however, there may be clients with valid tickets to that principal that are encoded with the old key. To save clients the trouble of having to request new tickets to a noninteractive principal when the principal's key has changed, every key is flagged with a version number, and old key versions are retained until all tickets that could have been encoded with that key have expired.

Finally, if a noninteractive principal's key has been compromised, it may be invalidated (along with all the corresponding tickets held by any clients) by simply deleting it from the local key storage.

Note: The key management API is for use only by applications using the DCE shared-secret authentication protocol and the key-type DES.

Retrieving a Key

The key management API provides two functions for retrieving a key from the local key storage. The **sec_key_mgmt_get_key()** function returns a specified key version for a specified principal. The meaning of specifying version 0 (zero) in this routine may vary depending on the authentication protocol in effect. (If the protocol is DCE shared-secret, the value 0 for the version identifier means the version that was most recently added to the local storage.) In any case, a principal's login is almost always successful if the principal uses the version 0 key.

When there are valid tickets that are encoded with different key versions, an application may need to retrieve more than one key version. In that case, the application may call **sec_key_mgmt_initialize_cursor()** to set a cursor in the local storage to the first suitable entry corresponding to the named principal and key type, and then call **sec_key_mgmt_get_next_key()** to get all versions of that key in storage. The application may then call **sec_key_mgmt_release_cursor()**, which disposes of information associated with the cursor**.** Neither of the key-retrieval routines can return keys that have been explicitly deleted, or that have been garbage collected after expiring.

The two key-retrieval functions dynamically allocate the memory for the returned key(s). To enable the efficient allocation of memory, an application may call **sec_key_mgmt_free_key()**, which frees the memory occupied by the key and returns it to the allocation pool.

Changing a Key

The **sec_key_mgmt_change_key()** function communicates with the registry to change the principal's key to a specified string, and also places the new string in the local key storage. The *keydata* input argument for this call may be a new key that the application specifies or a random key returned by the **sec_key_mgmt_gen_rand_key()** routine. An application may call **sec_key_mgmt_get_next_kvno()** to determine the next key version number that should be assigned to the new key so that it may reference this key version when retrieving a key.

In some circumstances, a principal may need to change its key in the local key storage but not immediately update the registry database. For example, a database application may maintain replicas of a master database that are managed by servers running on different computers. If these servers all provide exactly the same service, it makes sense for them to share the same key (meaning that they share the same principal identity). This way, a user with a ticket to the principal can be directed to whichever server is least busy.

When the registry database obtains a new key for a principal, the authentication service can immediately begin issuing tickets to the principal that are encoded under the new key. However, suppose the master for a single-principal replicated service were to call **sec_key_mgmt_change_key()**, and a client presented a ticket encoded with the latest key to a replica that had not yet learned that key. In this case, the replica would refuse service, even though the ticket was valid. Therefore, if an application employs replicated servers that are also instances of a single principal identity, the application should do the following:

1. Generate a new key by calling **sec_key_mgmt_gen_rand_key()**. This routine simply returns a key to the calling process, without updating the registry or local storage.
2. Disseminate the new key to all replicas.
3. Cause the replicas to call **sec_key_mgmt_set_key()**. This call updates the local storage to the new key but does not update the registry database entry for the principal. (The key version specified in this routine must not be 0 [zero].) The replicas should notify the master when they have completed setting their local stores to the new key.
4. Cause the master to call **sec_key_mgmt_change_key()** (here again, the key version must not be 0) after all replicas have set the new key locally, thereby updating both the master's local storage and the registry database entry.

Of course, if the master and each replica has its own principal identity, each server may call **sec_key_mgmt_change_key()** without coordinating this activity with any others.

Automatic Key Management

It is sometimes convenient for a principal to be able to change its key on a schedule determined by the password expiration policy for that principal, rather than to rely on a network administrator to decide when this should be done. In this case, the application may call **sec_key_mgmt_manage_key()**. This function invokes **sec_key_mgmt_gen_rand_key()** shortly before the current key is due to expire, updates both the local key storage and the registry database entry with the new key, and then calls **sec_key_mgmt_garbage_collect()** to discard any obsolete keys. This function runs indefinitely; it will never return during normal operation and so should be invoked from a thread dedicated to key management. It is not intended for use by server principals that share the same key.

Deleting Expired Keys

In order to prevent service interruptions, the key management API does not immediately discard keys that have been replaced; instead, it maintains the keys, with a version number and key-type identifier, in the local key storage. However, after a key has been out of use for longer than the maximum life of a ticket to the principal, it is no longer possible that any client of that principal has a valid ticket encoded with that key. At this time, the key storage may have its garbage collected.

The **sec_key_mgmt_garbage_collect()** routine collects garbage in the local key storage by deleting all keys older than the maximum ticket lifetime for the cell. The *garbage_collect_time* argument, which is returned by **sec_key_mgmt_change_key()**, specifies when key-storage garbage is to be collected.

Deleting a Compromised Key

When a principal's key has been compromised, it should be deleted as soon as the damage has been discovered in order to prevent another party from masquerading as that principal. Two routines delete a principal's key:

- The **sec_key_mgmt_delete_key()** routine removes all key types having the specified key version identifier from the local key storage, thus invalidating all extant tickets encoded with that key.
- The **sec_key_mgmt_delete_key_type()** routine removes only a specified version of a specified key type.

If the compromised key is the current one, the application should first change the key with **sec_key_mgmt_change_key()**. It is not an error for a process to delete the current key as long as it is done after the login context has been established, but it may inconvenience legitimate clients of a service. The inconvenience may be justified, however, if the application data is sensitive.

Since an application may have no means to discover that its key has been compromised, the **rgy_edit** tool provides interfaces that call **sec_key_mgmt_delete_key()**, **sec_key_mgmt_change_key()**, and **sec_key_mgmt_gen_rand_key()** so that a network administrator, who is more likely to detect that a key has been compromised, may handle a security breach of this kind. As an alternative, the application may provide user interfaces to these routines.

Chapter 32. The Access Control List APIs

As a rule, DCE Security Service interfaces are local client-side APIs only. The access control list (ACL) facility includes this kind of interface, and some others as well, as follows:

- The DCE client ACL interface, **sec_acl_ *()**, is a local interface that calls a client-side implementation of the ACL network interface. It enables clients to browse or edit DCE ACLs.
- The DCE server ACL manager library, **dce_acl_ *()**, enables servers to perform DCE-conformant authorization checks at runtime. This ACL library provides an implementation of the ACL manager interface and the ACL network interface. It supports the development of ACL managers for DCE servers.
- The DCE ACL network interface, **rdacl_ *()**, enables servers that manage access control to communicate with **sec_acl**-based clients.

Figure 80 provides a schematic view of the relationships and usage of these interfaces, as well as some relevant RPC interfaces. This chapter first discusses the client API, and then the two server program interfaces.

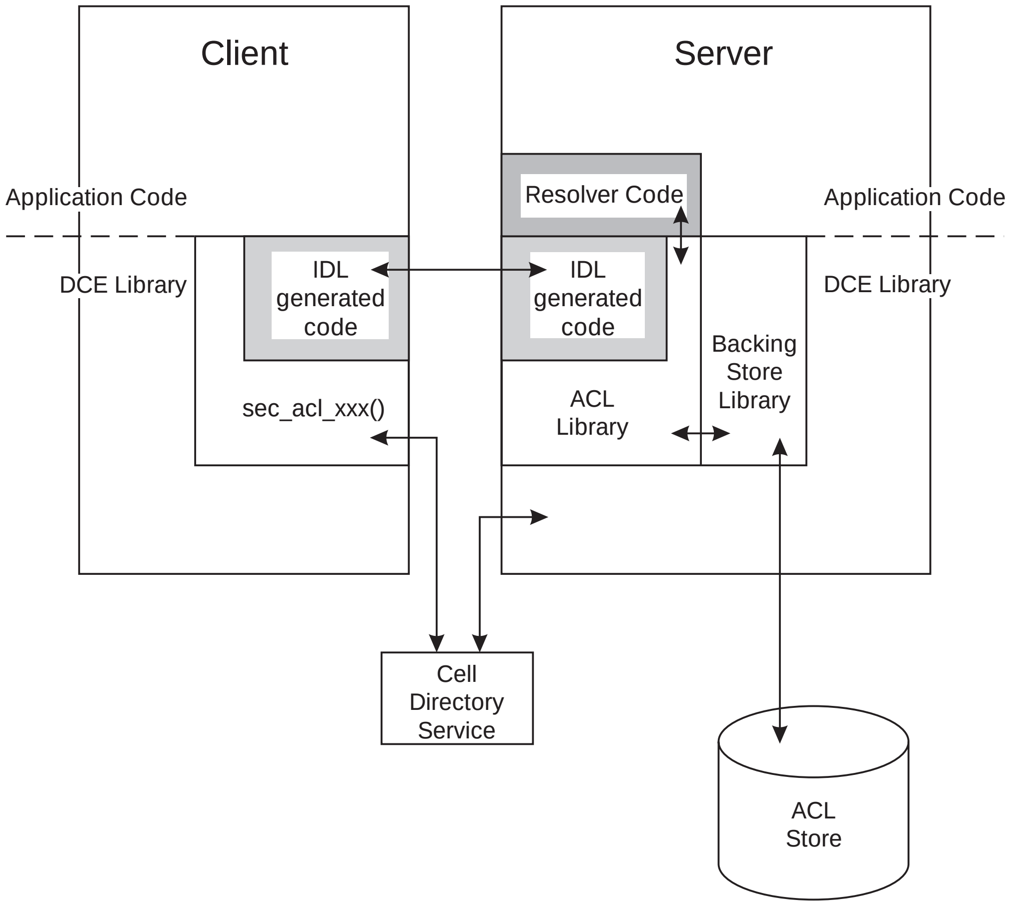


Figure 80. ACL Program Interfaces

The Client-Side API

The client-side API is a local interface consisting of a set of routines that are prefixed **sec_acl**. This is the interface on which the default DCE ACL editor (the DCE control program, or **dcecp**) is built. An application that needs to replace **dcecp** with a DCE ACL editor or browser of its own calls this interface. The following subsections provide specific information on the functionality that this API supports.

Binding to an ACL

Any operation performed on an ACL uses an ACL handle of type **handle_t** to identify the target of the operation. The handle is bound to the server that manages the object protected by the ACL, not to the ACL itself. Since an object may be protected by more than one ACL manager type (see “Chapter 25. Authorization” on page 523), the ACL itself can only be uniquely identified by the ACL handle in combination with the manager type that manages it. ACL editing calls must also specify the ACL type to be read or otherwise manipulated (the object, default container, or default Object ACL types).

An application calls **sec_acl_bind()** to get an ACL handle. The handle itself is opaque to the calling program, which needs none of the information encoded in it to use the ACL interface. A program can obtain the list of ACL manager types protecting an object and pass this data, along with the ACL type identifier, to another client-side routine. The following two calls perform this function:

- **sec_acl_get_manager_types()** returns a list of UUIDs of the manager types.
- **sec_acl_get_manager_types_semantics()** returns UUIDs of the manager types, and also the POSIX semantics supported by each manager type. The output of this call is used by the **sec_acl_calc_mask()** routine when it calculates a new **mask_obj** mask.

In the absence of CDS, an application may call **sec_acl_bind_to_addr()**; this call binds to a network address rather than a cell namespace entry.

Once an application is finished using an ACL handle, it should call **sec_acl_release_handle()** to dispose of it.

ACL Editors and Browsers

After obtaining a handle to the object in question (and using **sec_acl_get_manager_types()** or **sec_acl_get_manager_types_semantics()** to determine the ACL manager types protecting the object), editors and browsers use the **sec_acl_lookup()** function to return a copy of an object’s ACL.

Once an object’s ACL is retrieved, the editor can call **sec_acl_get_printstring()** to receive instructions about how to display the permissions of the ACL in a human-readable form. This call returns a symbol or word for each permission (a character string), and also a bitmask, with a bit (or bits) set to encode the permission. In addition, the print string structure includes a short explanation of each permission.

An ACL cannot be modified in part. To change an ACL, an editor must read the entire ACL (the **sec_acl_t** structure), modify it, and replace it entirely by calling **sec_acl_replace()**. If the ACL manager supports the **mask_obj** mask type, you can use **sec_acl_calc_mask()** to calculate a new **sec_acl_e_type_mask_obj** entry type. This function is supported for POSIX compatibility only, for those applications that use **mask_obj** with its POSIX semantics. Accordingly, **sec_acl_calc_mask()** returns the union of the permissions of all ACL entries other than **user_obj**, **other_obj**, **unauthenticated** (and the pre-existing **mask_obj**). These correspond approximately to what POSIX calls the "File Group Class" of ACL entries, although that designation is not appropriate in the DCE context. In particular, **sec_acl_calc_mask()** works independently of DCE DFS.

Use the **sec_acl_get_manager_types_semantics()** routine to obtain the required POSIX semantics and determine if the manager to which the ACL list will be submitted supports the **sec_acl_e_type_mask_obj** entry type.

An ACL can occupy a substantial amount of memory. The memory management routine, **sec_acl_release()**, frees the memory occupied by an ACL, and returns it to the pool. This is implemented strictly as a local operation.

Errors

Although the ACL API saves errors received from the DCE RPC runtime (or other APIs) in ACL handle data, it returns an error describing the ACL operation that failed as a result of the RPC error. However, if an error occurs and the client needs to know the cause of the ACL operation failure, it may call **sec_acl_get_error_info()**. This routine returns the error code last stored in the handle.

Guidelines for Constructing ACL Managers

ACL manager names for all of DCE should follow the convention for naming **dcecp** attributes. There is no architectural restriction involved in the guidelines shown here, merely an attempt at consistency. The DCE control program will accept names outside of this convention, but adherence to it will make usage of ACL managers easier.

The guidelines are as follows:

- Alphabetic characters in names must be lowercase only.
- Names should not contain underscores.
- Names should not contain spaces.
- Names should be no longer than 16 bytes, the defined value of **sec_acl_printstring_len**.
- Names should be similar to object command names supported in **dcecp** whenever possible. For example, the ACL manager name **principal** refers to the object, **/.:/sec/principal**, that contains registry information about principals. Note that **dcecp** allows abbreviations. For example, a user can specify **org** for the ACL manager name **organization**.
- Names must be unique within a component's ACL manager but not necessarily within DCE. For example, the name **xattrschema** can be used for a DCE extended attribute configuration schema ACL object and for a security ERA schema ACL object.
- The help string for an ACL manager must specify the component that owns or manages the objects in question because this information cannot always be derived from the ACL manager name.

Extended Naming of Protected Objects

The DCE ACL model supports extended naming so that ACL managers can separately protect objects that are not registered in the cell namespace. This provides an alternative to registering all the server's objects with CDS. The server alone is registered, and it contains code to identify its own objects by name. To achieve ACL protection for these objects, the ACL manager must be able to identify the ACLs in the same way the server identifies the objects. A resolution routine provides this ability.

Figure 81 shows the example of a printer server that is registered with CDS, with printers that are not. The ACL manager for the printer server uses the **dce_acl_resolve_by_name()** resolution routine to obtain the UUIDs of the several printers that are supported. The administrator in charge of the printers can change the printers, their names, and their ACLs without concern for registering them with CDS.

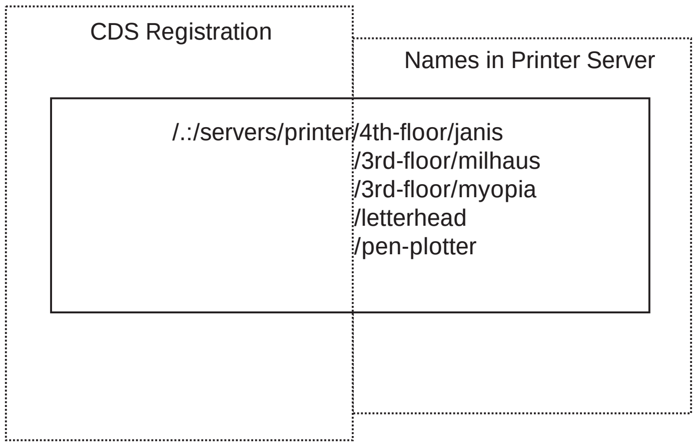


Figure 81. Protection with Extended Naming

When the **dce_acl_register_object_type()** routine registers an object type, it associates a resolution routine with the object type. The ACL library provides two resolution routines: **dce_acl_resolve_by_name()** and **dce_acl_resolve_by_uuid()**. Other resolution routines can be easily written, as required.

To take advantage of extended naming, an ACL manager must register the server name, object UUID, and **rdaclif.idl** interface with the CDS. (Refer to the *OSF DCE Application Development Guide—Directory Services* for more information). In addition, the ACL manager must register the object UUID and **rdaclif.idl** interface with the RPC endpoint mapper (refer to the chapters concerning RPC in "Part 3. DCE Remote Procedure Call" on page 147 of this guide).

The ACL Network Interface

The ACL network interface, **rdacl_ *()**, provides a DCE-common interface to ACL managers. It is the interface exported by the default DCE ACL managers to the default DCE ACL client (that is, the **dcecp** tool), and any other client based on the client API.

The client API, **sec_acl_ *()**, is a local interface that calls a client-side implementation of the ACL network interface. The server side implementation of this interface must conform to the **rdacl_ *(3sec)** reference pages. The DCE ACL library provides such an implementation. Following is a summary of the **rdacl_ *()** routines:

- **rdacl_lookup()**

  Retrieves a copy of the object's ACL.
- **rdacl_replace()**

  Replaces the specified ACL.
- **rdacl_get_access()**

  Returns a principal's permissions to an object (useful for implementing operations like the conventional UNIX system access function).
- **rdacl_test_access()**

  Determines whether the calling principal has the requested permission(s).
- **rdacl_test_access_on_behalf()**

  Determines whether the principal represented by the calling principal has the requested permission(s). This function returns TRUE if both the principal and the calling principal acting as its agent have the requested permission(s).

  Note: The **rdacl_test_access_on_behalf()** routine is deprecated and should not be used in new code. Delegation has removed the need for this routine.
- **rdacl_get_manager_types()**

  Returns a list of manager types protecting the object.
- **rdacl_get_printstring()**

  Obtains human-readable representations of permissions.
- **rdacl_get_referral()**

  Returns a referral to an ACL update site. This function enables a client that attempts to modify an ACL at a read-only site to recover from the error and rebind to an update site.

The ACL Library

The ACL library provides an implementation of the ACL manager interface and the ACL network interface for the convenience of programmers who are writing ACL managers for DCE servers.

The ACL library meets the following needs:

- It provides stable storage for ACLs.
- It implements the **rdacl_ *()** interface, including support for multiple object types, initial default Object ACLs, and initial default Container ACLs.
- It implements the full access algorithm, including masks and delegation.
- It provides DCE developers with a set of convenience functions so that servers can easily perform common styles of access control with minimal effort.

ACL Library Capabilities

The ACL library provides simple and practical access to the DCE security model.

The library provides a routine that indicates in a single call whether or not a client has the appropriate permissions to perform a particular operation. A server can also easily retrieve the full set of permissions granted to a client by an object's ACL.

The library provides the complete **rdacl_ *()** remote interface. Standard routines are provided to map either a UUID attached to a handle or a residual name specified as one of the parameters.

The combination of these capabilities means that most servers will not have any need to use DCE ACL data types directly.

The ACL API

The ACL library API, **dce_acl_ *()**, is a local interface that provides the server-side implementation of the ACL network interface. The reference pages in *OSF DCE Application Development Reference* describe the library routines.

The ACL library consists of the following parts:

- Initialization routines, where the server registers each ACL manager type.
- Server queries, where a server can perform various types of access checks.
- ACL object creation, where servers can create ACLs without concern for most low-level data type details.
- The **rdacl_ *()** implementation and server callback, where the server maps **rdacl_ *()** parameters into a specific ACL object. Two sample resolver routines are associated with this part:
  – **dce_acl_resolve_by_name()**

    Finds an ACL's UUID, given an object's name.

  – **dce_acl_resolve_by_uuid()**

    Finds an ACL's UUID, given an object's UUID.

Initialization Routines: An ACL manager must first define the types of the objects it manages. For example, a simple directory service would have directories and entries, and each type of object would have a different ACL manager. On a practical level, if a server has different types of objects, then the most common difference between the ACL managers is the printed representation of its permission bits. In other words, although the **sec_acl_printstring_t** values differ, the algorithm for evaluating permissions remains the same.

The ACL library provides a global print string that specifies the **read**, **write**, and **control** bits. Application developers are encouraged to use this print string whenever appropriate.

An ACL manager calls the **dce_acl_register_object_type()** routine to register an object type, once for each type of object that the server manages. The manager print string does not define any permission bits; they are set by the library to be the union of all permissions in the ACL print string.

The server must register the **rdacl_ *()** interface with the RPC runtime and with the endpoint mapper. See the **dce_server_register(3dce)** reference page.

Server Queries: The ACL library provides several routines to automate the most common use of DCE ACLs:

- **dce_acl_is_client_authorized()**

  Checks whether a client's credentials are authenticated and, if so, that they grant the desired access.
- **dce_acl_inq_client_permset()**

  Returns the client's permissions, corresponding to an ACL.

- **dce_acl_inq_client_creds()**
  Returns the client's credentials.
- **dce_acl_inq_permset_for_creds()**
  Determines a client's complete extent of access to an object.
- **dce_acl_inq_acl_from_header()**
  Retrieves the UUID of an ACL from the header of an object in the backing store.
- **dce_acl_inq_prin_and_group()**
  Inquires the principal and the group of an RPC caller.

Creating ACL Objects: The following convenience functions may be used by an application programmer to create ACL objects in other servers or clients.

- **dce_acl_copy_acl()**
  Copies an ACL.
- **dce_acl_obj_init()**
  Initializes an ACL for an object.
- **dce_acl_obj_free_entries()**
  Frees space used by an ACL's entries.
- **dce_acl_obj_add_user_entry()**
  Adds permissions for a user ACL entry to the given ACL.
- **dce_acl_obj_add_group_entry()**
  Adds permissions for a group ACL entry to the given ACL.
- **dce_acl_obj_add_id_entry()**
  Adds permissions for an ACL entry to the given ACL.
- **dce_acl_obj_add_unauth_entry()**
  Adds permissions for an **unauthenticated** ACL entry to the given ACL.
- **dce_acl_obj_add_obj_entry()**
  Adds permissions for an **obj** ACL entry to the given ACL.
- **dce_acl_obj_add_foreign_entry()**
  Adds permissions for the ACL entry for a foreign user or group to the given ACL.
- **dce_acl_obj_add_any_other_entry()**
  Adds permissions for the **any_other** ACL entry to a given ACL.

RDACL Implementation and Server Callback: The ACL library makes a complete implementation of the **rdacl_ *()** interface available to programmers writing servers, in a manner that is mostly transparent to the rest of the server code.

The operations in the **rdacl_ *()** interface share an initial set of parameters that specify the ACL object being operated upon:

```
handle_t            h
sec_acl_component_name_t    component_name
uuid_t             *manager_type
sec_acl_type_t         sec_acl_type
```

The *sec_acl_type* parameter indicates whether a protection ACL, an initial default Object ACL, or an initial default Container ACL is desired. It does not appear in the **access** operations as it must have the value **sec_acl_type_object**.

In order to implement the **rdacl_ *()** interface, the server must provide a **resolution** routine that maps these parameters into the UUID of the desired ACL object; the library includes two such routines: **dce_acl_resolve_by_uuid()** and **dce_acl_resolve_by_name()**.

The resolution routine is required because servers use the namespace in different ways. Here are three examples:

- Servers that export only their binding information and manage a single object, and hence use a single ACL, do not need the resolution parameters. DTS is an example of this case.
- Servers with many objects in the namespace, with a UUID in each entry, will call **rpc_binding_inq_object** on the handle to obtain the object UUID. They then use this same UUID as the index of the ACL object. Many application servers will be of this type. One ACL library resolver function, **dce_acl_resolve_by_uuid()**, matches this paradigm. This paradigm is not appropriate if the number of objects is immense.
- Servers with many objects will use a junction or similar architecture so that the component name (also called the **residual**) specifies the ACL object by name. The DCE security server is essentially of this type. Another ACL library resolver function, **dce_acl_resolve_by_name()**, matches this paradigm.

The following **typedef** specifies the signature for a resolution routine. The first four parameters are the common **rdacl_ *()** parameters mentioned previously.

```
typedef void (*dce_acl_resolve_func_t)(
/* [in] parameters */
  handle_t          h,
  sec_acl_component_name_t  component_name,
  sec_acl_type_t       sec_acl_type,
  uuid_t           *manager_type,
  boolean32          writing,
  void            *resolver_arg
/* [out] parameters */
  uuid_t           *acl_uuid,
  error_status_t       *st
);
```

For situations in which neither of the ACL library resolver functions, **dce_acl_resolve_by_uuid()** or **dce_acl_resolve_by_name()**, is appropriate, application developers must provide their own.

The following two examples illustrate the general structure of the **dce_acl_resolve_by_uuid()** API and **dce_acl_resolve_by_name()** API that are supplied in the ACL library. They may be used as paradigms for creating additional resolver routines.

The first example shows **dce_acl_resolve_by_name()**.

A server has several objects and stores each in a backing store database. Part of the standard header for each object is a structure that contains the UUID of the ACL for that object. (The standard header is not intended to be an abstract type, but rather a common prolog provided to ease server development.) The resolution routine for this server retrieves the object UUID from the handle, uses that as an index into its own backing store, and uses the *sec_acl_type* parameter to retrieve the appropriate ACL UUID from the standard data header.

This routine needs the database handle for the server's object storage, which is specified as the *resolver_arg* parameter in the **dce_acl_register_object_type()** call.

```
#define STAT_CHECK_RET(st) { if (st != error_status_ok) return; }
dce_acl_resolve_func_t
dce_acl_resolve_by_uuid(
 /* in */
  handle_t h,
  sec_acl_component_name_t component_name,
  sec_acl_type_t sec_acl_type,
  uuid_t *manager_type,
  boolean32 writing,
  void *resolver_arg,
 /* out */
  uuid_t *acl_uuid,
  error_status_t *st
)
{
  dce_db_handle_t       db_h;
  dce_db_header_t       dbh;
  uuid_t           obj;

  /* Get the object. */
  rpc_binding_inq_object(h, &obj, st);
  STAT_CHECK_RET(*st);
  /* Get object header using the object backing store.
   * The handle was passed in as the resolver_arg in the
   * dce_acl_register_object_type call.
   */
  db_h = (dce_db_handle_t)resolver_arg;
  dce_db_std_header_fetch(db_h, &obj, &dbh, st);
  STAT_CHECK_RET(*st);

  /* Get the appropriate ACL based on the ACL type. */
  dce_acl_inq_acl_from_header(dbh, sec_acl_type, acl_uuid, st);
  STAT_CHECK_RET(*st);
}
```

The next example shows **dce_acl_resolve_by_name()**.

A server uses the residual name to resolve an ACL object by using **dce_acl_resolve_by_name()**. This routine requires a DCE database that maps names into ACL UUIDs. This backing store database must be maintained by the server application so that created objects always get a name, and that name must be a key into a database that stores the UUID identifying the object. The *resolver_arg* parameter given in the **dce_acl_register_object_type()** call must be a handle for that database.

```
#define STAT_CHECK_RET(st) { if (st != error_status_ok) return; }
dce_acl_resolve_func_t
dce_acl_resolve_by_name(
 /* in */
  handle_t h,
  sec_acl_component_name_t component_name,
  sec_acl_type_t sec_acl_type,
  uuid_t *manager_type,
  boolean32 writing,
  void *resolver_arg,
 /* out */
  uuid_t *acl_uuid,
  error_status_t *st
)
{
  dce_db_handle_t       db_h;
  dce_db_header_t       dbh;
```

```
    /* Get object header using the object backing store.
     * The handle was passed in as the resolver_arg in the
     * dce_acl_register_object_type call.
     */
    db_h = (dce_db_handle_t)resolver_arg;
    dce_db_std_header_fetch(db_h, component_name, &dbh, st);
    STAT_CHECK_RET(*st);

    /* Get the appropriate ACL based on the ACL type. */
    dce_acl_inq_acl_from_header(dbh, sec_acl_type, acl_uuid, st);
    STAT_CHECK_RET(*st);

  }
```

Chapter 33. The ID Map API

In the multicell environment, the global print string representation of a principal identity can be ambiguous, even though every principal and its native cell have unique names in the form of UUIDs to which the print string representations normally resolve. For example, all ACLs maintain UUIDs as the definitive representations of principal and cell names. The **acl_edit** tool, on the other hand, takes as input (and also outputs) this same information as print strings. This string-to-UUID mapping is accomplished easily enough when an ACL entry refers to a local identity; that is, a member of the local cell. However, when a user adds an ACL entry for a foreign principal identity such as **/.../world/dce/rd/writers/tom**, it is not evident to the ACL manager which part of the name identifies the cell, and which identifies the principal within the cell. The name **/.../world/dce** may refer to a cell containing the principal **/rd/writers/tom**, or the cell name may be **/.../world/dce/rd** and the principal name **/writers/tom**.

To parse the fully qualified principal name that the user types into its cell name and local principal-name components, and for these components to be mapped to UUIDs, ACL managers that support entries for foreign identities use the ID map API. For the same reasons, many other kinds of servers in a DCE multicell environment need a facility to parse global names and translate UUIDs into print string names.

The ID map API provides a simple interface to translate a fully qualified name (that is, the global representation of a name) into its components and back again. This API consists of the following calls:

- The **sec_id_parse_name()** call takes as input a registry context handle and a fully qualified principal name, and returns the principal's print string name and UUID, and the print string name and UUID of the principal's native cell.
- The **sec_id_gen_name()** call translates a principal UUID and the UUID of its native cell UUID into a cell-relative principal name, a cell name, and a fully qualified principal name.
- The **sec_id_parse_group()** call is **likesec_id_parse_name()**, except that it operates on group names.
- The **sec_id_gen_group()** call is like **sec_id_gen_name()**, except that it operates on group names.

Chapter 34. DCE Audit Service

Audit plays a critical role in distributed systems. Adequate audit facilities are necessary for detecting and recording critical events in distributed applications.

Audit, a key component of DCE, is provided by the DCE Audit Service.

This chapter provides an introduction to the DCE Audit Service.

Features of the DCE Audit Service

The DCE Audit Service has the following features:

- An audit daemon performs the logging of audit records based on specified criteria.
- Application programming interfaces (APIs) can be used as part of application server programs to record audit events. These APIs can also be used to create tools that analyze the audit records.
- An administrative command interface to the audit daemon directs the daemon in selecting the events that are going to be recorded based on certain criteria.
- An event classification mechanism is used to logically group a set of audit events for ease of administration.
- Audit records can be directed to logs or to the console.

Components of the DCE Audit Service

The DCE Audit Service has three basic components:

- application programming interfaces (APIs)

  Provide the functions that are used to detect and record critical events when the application server services a client. The application programmer uses these functions at code points in the application server program to actuate the recording of audit events.

  Other APIs are also provided which can be used to create tools that examine and analyze the audit event records.
- audit daemon

  Maintains the filters and the audit logs.
- audit management interface

  Management interface to the audit daemon. Used by the administrator to specify how the audit daemon will filter the recording of audit events. This interface is available from the DCE control program.

DCE Audit Service Concepts

This section briefly describes the DCE Audit Service concepts that are relevant to DCE application programming.

Audit Clients

All RPC-based servers, such as DCE servers and user-written application servers, are potential audit clients. The DCE Security Service, DTS, and the DCE Audit

Service itself are auditable. That is, code points (discussed in “Code Point”) are already in place on these services.

The audit daemon can also audit itself.

Code Point

A code point is a location in the application server program where DCE audit APIs are used. Code points generally correspond to operations or functions offered by the application server for which audit is required. For example, if a bank server offers the cash withdrawal function **acct_withdraw()**, this function may be deemed to be an auditable event and be designated as a code point.

As mentioned previously, code points are already in place in the DCE Security Service, DTS, and DCE Audit Service. Code points and their associated events for the DCE Security Service are documented in the **sec_audit_events(5sec)** reference page. Code points and their associated events for the DTS are documented in the **dts_audit_events(5sec)** reference page. Code points and their associated events for the DCE Audit Service are documented in the **aud_audit_events(5sec)** reference page.

Events

An audit event is any event that an audit client wishes to record. Generally, audit events involve the integrity of the system. For example, when a client withdraws cash from his bank account, this can be an audit event.

An audit event is associated with a code point in the application server code.

The terms audit event, event, and auditable event are used interchangeably in this book.

Event Names and Event Numbers

Each event has a symbolic name as well as a 32-bit number assigned to it. Symbolic names are used only for documentation in identifying audit events. In creating event classes, the administrator uses the event numbers associated with these events.

Event numbers are 32-bit integers. Each event number is a tuple made up of a *set-id* and the *event-id*. The *set-id* corresponds to a set of event numbers and is assigned by OSF to an organization or vendor. The *event-id* identifies an event within the set of events. The organization or vendor manages the issuance of the event ID numbers to generate an event number.

Event numbers must be consecutive. That is, within a range of event numbers, no gaps in the consecutive order of the numbers are allowed.

The structure and administration of event numbers can be likened to the structure and administration of IP addresses. Recall that an IP address is a tuple of a network ID (analogous to the set-id) and a host ID (analogous to the event-id). The format and administration of event numbers are also analogous to IP addresses, as will be discussed in the next sections.

Event Number Formats

Events numbers follow one of five formats (A to E), depending on the number of audit events in the organization. The format of an event number can be determined from its four high-order bits.

Format A can be used by large organizations (such as OSF or major DCE vendors) that need more than 16 bits for the event-id. This format allocates 7 bits to the set-id and 24 bits to the event-id. Format A event numbers with zero (0) as its set-id are assigned to OSF. That is, all event numbers used by OSF have a zero in the most significant byte.

Format B can be used by intermediate-sized organizations that need 8 to 16 bits for the event-id.

Format C can be used by small organizations that need less than 8 bits for the event-id.

Format D is not administered by OSF and can be used freely within the cell. These event numbers may not be unique across cells and should not be used by application servers that are installed in more than one cell.

Format E is reserved for future use.

The event number formats are illustrated in Figure 82.

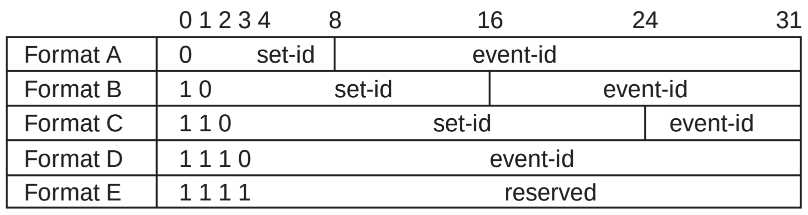


Figure 82. Event Number Formats

Sample Event Numbers for DCE Servers

Following are examples of event numbers in the security and time servers, as defined in a header file used by the security server and time server programs, respectively.

```
/* Event numbers 0x00000100 to 0x000001FF are assigned to the
  security server. */

#define AS_Request    0x00000100
#define TGS_TicketReq   0x00000101
#define TGS_RenewReq   0x00000102
#define TGS_ValidateReq  0x00000103
...
/* Event numbers 0x00000200 to 0x000002FF are
  assigned to the time server. */

#define CNTRL_Create   0x00000200
#define CNTRL_Delete   0x00000201
#define CNTRL_Enable   0x00000202
#define CNTRL_Disable   0x00000203
...
```

Sample Event Numbers for Application Servers

The following is an example of the event numbers in a banking server application, as defined in the application's header file.

```
#define evt_vn_bank_server_acct_open 0x01000000
#define evt_vn_bank_server_acct_close 0x01000001
#define evt_vn_bank_server_acct_withdraw 0x01000002
#define evt_vn_bank_server_acct_deposit 0x01000003
#define evt_vn_bank_server_acct_transfer 0x01000004
```

Administration of Event Numbers

Organizations and vendors must administer the event numbers assigned to them (through the set-id) to maintain the unique assignment of event numbers.

Event Class

Audit events can be logically grouped together into an event class. Event classes provide an efficient mechanism by which sets of events can be specified by a single value. Generally, an event class consists of audit events with some commonality. For example, in a bank server program, the cash transaction events (deposit, withdrawal, and transfer) may be grouped into an event class.

Typically, the administrator creates and maintains event classes. For more details to event classes, see the *OSF DCE Administration Guide—Core Components*.

Event Class Number

Each event class is assigned an event class number. Like the event number, the event class number is a 32-bit integer and is administered by OSF. Event class numbers are discussed in more detail in the *OSF DCE Administration Guide—Core Components*.

Filters

Once the code points are identified and placed in the application server, all audit events corresponding to the code points will be logged in the audit trail file, irrespective of the outcome of these audit events. However, recording all audit events under all conditions may neither be practical nor necessary. Filters provide a means by which audit records are logged only when certain conditions are satisfied. A filter is composed of filter guides that specify these conditions. Filter guides also specify what action to take if the condition (outcome) is met.

A filter answers the following questions:
- Who will be audited?
- What events will be audited?
- What should be the outcome of these events before an audit record is written?
- Will the audit record be logged in the audit trail file or displayed on the system console, or both?

For example, for the bank server program, you can impose the following conditions before an audit record is written:

"Audit all withdrawal transactions (the audit events) that fail because of access denial (outcome of the event) that are performed by all customers in the DCE cell (who to audit)."

Filter Subject Identity

A filter is associated with one filter subject, which denotes to what the filter applies. The filter subject is the client of the distributed application who caused the event to happen.

For more information on the filter subject identity, see the *OSF DCE Administration Guide—Core Components*.

Audit Records

An audit record has a header and a trailer. The header contains the common information of all events; for example, the identities of the client and the server, group privileges used, address, and time. The trailer contains event-specific information; for example, the dollar amount of a fund-transfer event.

Audit records are initialized and filled by calling the audit API functions.

There are four stages in the writing of an audit record:

1. First, the code point registers an audit event. At this point, the audit record does not yet have any form.
2. The audit record descriptor is built. This is a representation of the audit data that is built by the **dce_aud_start()**, **dce_aud_put_ev_info()**, and **dce_aud_commit()** functions. This is stored in a data structure in the client's core memory until the **dce_aud_commit()** function is called. This data is not IDL-encoded until the **dce_aud_commit()** call.
3. The audit record is written to the log. This is stored as IDL-encoded data in the audit log.
4. The audit record is transformed into human-readable form. This is a representation built in a data structure in the core memory by calls to the **dce_aud_next()** and **dce_aud_print()** functions. This is not an IDL-encoded representation.

Audit Trail File

The audit trail file contains all the audit records that are written by the audit daemon or the audit APIs. You can specify either a central audit trail file or a local audit trail file. The central audit trail file is maintained by the audit daemon. The local audit trail file is maintained by the audit library. The terms audit trail file and audit trail are used interchangeably in this book.

Administration and Programming in DCE Audit

This section gives you an example of how auditing is accomplished using the DCE Audit Service. Both the programmer and the administrator have to perform tasks to enable the writing of audit records in the audit trail. This section looks at the life cycle of an audit trail, from the time that audit events are identified in the server code, to the time that they are filtered and recorded in the audit trail file.

A bank server example illustrates each stage of the life cycle. In this example, the bank server program offers five operations: **acct_open()**, **acct_close()**, **acct_withdraw()**, **acct_deposit()**, and **acct_transfer()**.

Programmer Tasks

The programmer uses the audit APIs to enable auditing in the application server program, as illustrated in the following:

1. The programmer identifies the code points in the bank server program. Because each of the five operations (corresponding to an RPC interface) offered by the bank server is a security-relevant operation, the programmer deems that all these operations are security relevant, and assigns a codepoint to each operation. Each code point corresponds to an audit event.

   ```
   acct_open()     /* first code point */
   acct_close()    /* second code point */
   acct_withdraw()   /* third code point */
   acct_deposit()   /* fourth code point */
   acct_transfer()   /* fifth code point */
   ```
2. The programmer then assigns an event number to each audit event (corresponding to each code point). For example, the programmer defines these numbers in his header file as follows:

   ```
   /* event number for the 1st code point, acct_open() */
   #define evt_vn_bank_server_acct_open   0xC1000000

   /* event number for the 2nd code point, acct_close() */
   #define evt_vn_bank_server_acct_close   0xC1000001

   /* event number for the 3rd code point, acct_withdraw() */
   #define evt_vn_bank_server_acct_withdraw 0xC1000002

   /* event number for the 4th code point, acct_deposit() */
   #define evt_vn_bank_server_acct_deposit  0xC1000003

   /* event number for the 5th code point, acct_transfer() */
   #define evt_vn_bank_server_acct_transfer  0xC1000004
   ```
3. The programmer now starts adding audit API functions to the bank server program.

   In the initialization part of the server, the application programmer uses the **dce_aud_open()** API to open an audit trail file for writing the audit records. This function uses the lowest-numbered event as one of its parameters; in this case, **0xC1000000** (**evt_vn_bank_server_acct_open**). Using the lowest-numbered event enhances the performance of the filter search.

   ```
   /* open an audit trail file for writing */
   dce_aud_open(aud_c_trl_open_write, description,
           evt_vn_bank_server_acct_open,
           5, &audit_trail, &status);
   ```
4. The programmer invokes the following DCE audit APIs at each code point:
   - The **dce_aud_start()** API, to initialize an audit record. This function assigns the event number to the event represented by the code point. Thus, it uses the event number corresponding to that code point as one of its parameters.
   - The **dce_aud_put_ev_info()** API, to add event-specific information to the audit record.
   - The **dce_aud_commit()** API, to commit the audit record in the audit trail file.

The use of these three APIs is illustrated in the following example of the bank server program:

```
acct_open()   /* first code point */

/* Uses the event number for acct_open(),
  evt_vn_bank_server_acct_open */
dce_aud_start(evt_vn_bank_server_acct_open,
       binding,options,outcome,&ard, &status);

/* If events need to be logged,
  add trailer info (optional) */
if (ard)
 dce_aud_put_ev_info(ard,info,&status);

/* If events need to be logged,
  add header and trailer info */
if (ard)
 dce_aud_commit(at,ard,options,format,&outcome,&status);

acct_close()  /* second code point */

/* Uses the event number for acct_close(),
 * evt_vn_bank_server_acct_close */
dce_aud_start(evt_vn_bank_server_acct_close,
       binding,options,outcome,&ard, &status);

if (ard) /* If events need to be logged */
 dce_aud_put_ev_info(ard,info,&status);

if (ard) /* If events need to be logged */

dce_aud_commit(at,ard,options,format,&outcome,&status);
```

5. The programmer uses the **dce_aud_close()** API in the termination routine of the application server. This API closes the audit trail file (and frees up memory) if the applicaton server shuts down.

   The coding of the application program to enable auditing is essentially complete at this point.

Administrator Tasks

The following steps will be performed by the administrator to filter the audit events and control the audit trail file.

1. The administrator obtains the event numbers corresponding to the events represented by the code points in the bank server program from the programmer or from the program’s documentation. These events and their assigned event numbers are as follows:

acct_open()	0xC1000000
acct_close()	0xC1000001
acct_withdraw()	0xC1000002
acct_deposit()	0xC1000003
acct_transfer()	0xC1000004

2. The administrator decides to create two event classes: the **account_creation_operations** class comprised of **acct_open()** and **acct_close()**, and the **account_balance_operations** class comprised of **acct_withdraw()**, **acct_deposit()**, and **acct_transfer()**.
3. The administrator decides to create two filters: one for all users within the cell (for the cell **/.:/torolabcell**), and the other for all other users.

   The filter for all users within the cell has the following guides:

   a. Audit the events in the event class **account_balance_operations** only, subject to the next condition.
   b. Write an audit record only if an operation in that event class failed because of access denial.
   c. If the first condition is fulfilled, write the audit record in an audit trail file only.

   The filter for all other users has the following filter guides:

   a. Audit the events in both event classes, subject to the next condition.
   b. Write an audit record if an operation in that event class succeeded or failed.
   c. Write the audit record both in an audit trail file and the console.

The scenarios described here can be summarized as follows:

- The programmer identifies the code points in the distributed application corresponding to the audit events.
- The programmer uses the audit API functions on those code points to enable auditing.
- The administrator creates event classes that are used to group the audit events.
- The administrator creates filters to narrow down the conditions by which audit records are written for the audit events.

Figure 83 illustrates the interactions among the audit client program, the audit API functions (**libaudit**), the audit daemon (**auditd**), and the audit management interface (available from the DCE control program, **dcecp**).

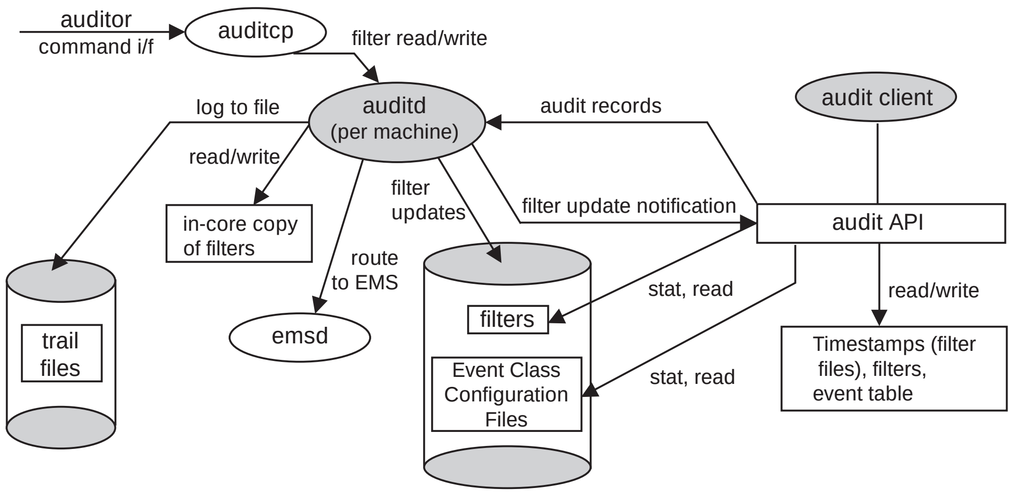


Figure 83. Overview of the DCE Audit Service

The audit management interface (accessed through the DCE control program) is used by the systems administrator to specify who, what, when, and how to audit. This is accomplished through the use of the filters. The audit daemon maintains the filter’s information in its address space. The filters are also stored in local files so that the filters can be restored when the machine restarts, and so that audit clients can read the filter information from these files.

The audit clients are the users of the filter information. Using the audit APIs, the audit client reads the information on filters and event class configuration. The audit client reads these files only once, unless an update notification is received from the audit daemon (which is triggered by an update initiated by an administrator using the DCE control program).

Chapter 35. Using the Audit API Functions

This chapter describes the use of the audit API functions to add audit capability to distributed applications and to write audit trail analysis and examination tools.

Adding Audit Capability to Distributed Applications

To record audit events in an audit trail file, the DCE audit API functions must be called in the distributed application to perform the following:

1. Open the audit trail file during the startup of the application.
2. Initialize the audit records at each code point.
3. Add event information to the audit records at each code point. (This is optional.)
4. Commit the audit records at each code point.
5. Close the audit trail file when the application shuts down.

Note that steps 2, 3, and 4 are repeated in sequence at each code point in the distributed applcation.

The use of the audit API functions in each of these steps is illustrated with the bank server example introduced in the previous chapter.

Five code points are identified in the bank server program: **acct_open()**, **acct_close()**, **acct_withdraw()**, **acct_deposit()**, and **acct_transfer()**. Each code point has been assigned an event number and defined in the application server's header file as follows:

```
#define evt_vn_bank_server_acct_open 0x01000000
#define evt_vn_bank_server_acct_close 0x01000001
#define evt_vn_bank_server_acct_withdraw 0x01000002
#define evt_vn_bank_server_acct_deposit 0x01000003
#define evt_vn_bank_server_acct_transfer 0x01000004
```

Opening the Audit Trail

To open the audit trail file, the main routine of the application server uses the **dce_aud_open()** function. With this function call, the audit trail file can be

- opened for reading or for writing.
- directed to the default audit trail file or to a specific file. If **dce_aud_open()** is called without specifying an audit trail file, (by having NULL as the value of the *description* parameter), a default audit trail file is used. This is the central trail file that is accessed by RPC calls to the audit daemon.

  If an audit trail file is specified in the **dce_aud_open()** call, (through the *description* parameter), that file is opened directly by the audit library, bypassing RPCs and the audit daemon.

In the bank server application, the function call is as follows:

```
dce_aud_open(aud_c_trl_open_write, &audit_file,
    evt_vn_bank_server_acct_open,
    5, &audit_trail, &status);
```

In this call, the audit trail file **audit_file** is opened for writing. The third parameter (**evt_vn_bank_server_acct_open**) specifies the lowest event number used in the bank server application. The fourth parameter (**5**) specifies the number of events defined.

The call returns an audit-trail descriptor (**audit_trail**) that will be used to append audit records to the audit trail file.

Initializing the Audit Records

Audit records can be initialized by using the **dce_aud_start_ *()** functions. This function has five variations, and the use of each variation depends on the available information about the server. In general, if you have the RPC binding information about the server, use the **dce_aud_start()** function. If not, use the other four variations of this function, depending on the available information. The five variations are as follows:

- **dce_aud_start()**

  For use by DCE RPC-based server applications.
- **dce_aud_start_with_server_binding()**

  For use by DCE RPC-based client applications.
- **dce_aud_start_with_pac()**

  For use by applications that do not use DCE RPC, but use the DCE authorization model.
- **dce_aud_start_with_name()**

  For use by applications that use neither DCE RPC nor the DCE authorization model.
- **dce_aud_start_with_uuid()**

  For use by RPC-based applications that know their client’s identity in UUID form.

The **dce_aud_start_ *()** functions determine if a specified event must be audited based on the subject identity and event outcome that were defined for that event by the filters.

If the event specifics match the event filters (that is, the event has to be audited), these functions return a pointer to an audit record buffer. If it is determined that the event does not need to be audited, a NULL pointer is returned, and the application can then discontinue any auditing activity. If it cannot be determined whether the event needs to be audited (because the event needs to be audited based on a specific outcome(s) but the outcome is not yet known) these functions return a non-NULL pointer.

When an audit record is initialized, the identification of the audit subject (that is, the client of the distributed application) is recorded.

You can use the **dce_aud_start_ *()** functions to specify the amount of header information in the audit record. You can specify any or acombination of the following:

- Information on all groups and addresses
- Information on groups only
- Information on addresses only.

Using these functions, you can bypass the filter altogether and log the event to the audit trail file or display it on the system console. This option is useful for applications whose events require unconditional audit actions.

In our example, each of the bank server routines (**acct_open()**, **acct_close()**, **acct_withdraw()**, **acct_deposit()**, **acct_transfer()**) will make a **dce_aud_start()** function call. In the **acct_transfer()** routine, the function call is made as follows:

```
acct_transfer()

dce_aud_start (evt_vn_bank_server_acct_transfer,
    h, aud_c_evt_all_info,
    aud_c_esl_cond_success, &ard, &status);
```

where **h** points to the RPC binding of the client making the call. The **aud_c_evt_all_info** option means that all information about the client's groups and addresses are included in the audit record header. The **aud_c_esl_cond_success** event outcome means that the event completed successfully.

Adding Event-Specific Information

If the **dce_aud_start()** function returns an audit record descriptor to the audit record buffer (meaning that the event needs to be audited), the **dce_aud_put_ev_info()** function call can be used to add event-specific information to the tail of the audit record.

You can opt not to use the **dce_aud_put_ev_info()** function if the information provided by the audit record header is already sufficient for your auditing purposes.

If you elect to use this function, it can be called one or more times, the order of which is preserved in the audit record.

The **dce_aud_put_ev_info()** function has two parameters: the *ard* parameter, which is the pointer to the audit record descriptor, and the *info* parameter, which is a **dce_aud_ev_info_t** type data containing the event-specific information. The programmer can specify the **dce_aud_ev_info_t** data type to include all the audit information that needs to be collected. For more information on the formats of the audit record, see the *OSF DCE Application Development Reference*.

In the **acct_transfer()** code point of the bank server example, if you want to record the account numbers of the parties involved in the transfer and the amount of each transaction, the data type declarations and the function calls can be made as follows:

```
dce_aud_ev_info_t info;

/* account numbers and transfer amounts are all unsigned
  32-bit integers */
info.format = aud_c_evt_info_ulong_int;

info.data = acct_from;
dce_aud_put_ev_info(ard, info, &status);
info.data = acct_to;
dce_aud_put_ev_info(ard, info, &status);
info.data = amount;
dce_aud_put_ev_info(ard, info, &status);
```

Committing an Audit Record

After the header and the optional tail information has been included in the audit record, the **dce_aud_commit()** function call is used to write the audit record in the audit trail file. This function uses the audit trail file previously opened by the **dce_aud_open()** function.

You can specify one of two options in the way the function writes the audit record in the audit trail file:

- Return an error status if the storage or logging service is not available when an attempt is made to write the audit record. This option can be used if the application program can handle write failures in the stable storage.
- If the storage or logging service is not available, keep on trying until the function is able to write to it. This option can be used if the audit record must be written to stable storage before the routine can proceed safely to another task.

In the bank server example, the function call can be made as follows:

```
dce_aud_commit(audit_trail, ard, options, format, outcome, &status);
```

The **audit_trail** parameter is the trail descriptor returned from the **dce_aud_open()** call made earlier. The **ard** parameter is the audit record descriptor returned from the **dce_aud_start()** call (and used in the **dce_aud_put_ev_info()** function call). The *format* parameter specifies a format version number of the event-specific information. The initial version number should be zero, and be incremented when the format changes. For example, the data type used for account numbers might change from 32-bit integer to UUID. The event outcome must be provided in this call, even if it has been provided in the **dce_aud_start()** call made earlier. If the event outcome (except **aud_c_esl_cond_unknown**) is provided in both calls, the values must be the same.

Closing an Audit Trail File

The audit trail file must be closed using the **dce_aud_close()** function when the application shuts down (because of the **rpc_mgmt_stop_server_listening()** function call or other exceptional conditions). For example, to close the trail, the bank server's main program can make the following function call:

```
dce_aud_close(audit_trail, &status);
```

This function flushes buffered audit records to stable storage and releases the memory allocated for the trail descriptor.

Writing Audit Trail Analysis and Examination Tools

The audit APIs can be used to write audit trail analysis and examination tools that selectively review the following:

- Events that are invoked by one or more subjects, for example, principals, groups, and cells
- Events that have a specific outcome
- Events that occurred during a specified time period
- Events that have specific event IDs

In its most basic form, an audit trail analysis and examination tool must perform five functions:

- Open an audit trail file for reading
- Read the audit records into a buffer
- Transform the audit records into human-readable form
- Discard the audit record
- Close the audit trail file

These functions and the APIs that are used for each are discussed in the following sections.

Opening an Audit Trail File for Reading

To open the audit trail file for reading, use the **dce_aud_open()** function and specify **aud_c_trl_open_read** as the value for the *flags* parameter. In this case, the values for the *first_evt_number* and *num_of_evts* does not affect the call. For example:

```
dce_aud_open(aud_c_trl_open_read, AUDIT_TRAIL_FILE,
    0, 0, &out_trail, status);
```

Reading the Desired Audit Records into a Buffer

After opening the audit trail file, you can use the **dce_aud_next()** function to retrieve audit records. Audit records are stored in the audit trail file in binary form. The **dce_aud_next()** function does not convert the file into readable form. You must use the **dce_aud_print()** function to translate the audit record into readable form.

The **dce_aud_next()** function allows you to specify a criteria that will be used in selecting the records that will be read from the file. This criteria is known as predicates and is expressed by setting the condition on the value of certain attributes. The condition is set by using any of the following operators: **=** (equal to), **>** (greater than), and **<** (less than).

Predicates can be expressed in any of the following forms:

- *attribute*= *value*
- *attribute*> *value*
- *attribute*< *value*

The following list summarizes these attributes and their acceptable values:

SERVER	UUID of the principal that generated the record
EVENT	Audit event number
OUTCOME	Event outcome of the record
STATUS	Authorization status of the application client
CLIENT	UUID of the client principal
TIME	Time when the record was generated.

CELL The UUID of the application client's cell

GROUP
The UUID of the application client's group or groups

ADDR The address (binding handle) of the client

FORMAT
The format version number of the audit event record

Details of these attributes, their values, and the allowable operators are discussed in the *OSF DCE Application Development Reference*.

For example, to have the function retrieve audit records that pertain to the event number 0xC01000001 only, you can set the predicate to the following:

```
EVENT=0xC01000001
```

If the predicate parameter is set to NULL (that is, no criteria), the next audit record is read. For example, to read the next audit record in a previously opened audit trail file, the following call is made:

```
dce_aud_next(out_trail, NULL, &out_ard, status);
```

You can specify multiple predicates, in which case the predicates are treated as a logical AND condition.

The **dce_aud_next()** function returns a pointer to the record that was read. This pointer is used by the **dce_aud_print()**, **dce_aud_get_ev_info()**, and **dce_aud_get_header()** functions in transforming the audit records into ASCII format.

Transforming the Audit Record into Readable Text

After reading in the desired audit record by using the **dce_aud_next()** function, these binary audit records must be transformed into human-readable form.

You can use any of the following three functions to transform the audit record information to human readable form:

- **dce_aud_print()**
  Formats the entire audit record (header and tail) into ASCII format.
- **dce_aud_get_header()**
  Obtains the header information of the audit record and formats it into human readable form.
- **dce_aud_get_ev_info()**
  Obtains the event-specific information in the tail of the audit record and formats it into human readable form.

The **dce_aud_next()** function returns the address of the audit record to these functions. These functions then allocate memory for the ASCII-format buffer (using **malloc()**) and fills it with the ASCII representation of the audit record. The user must explicitly release this memory (using **free()**) when all audit record retrieving and transforming tasks have been accomplished.

Discarding the Audit Record

The **dce_aud_discard()** function frees the memory allocated to the binary version of the audit record, that is, the structure returned by the **dce_aud_next()** function. The **dce_aud_discard()** function does not free the structures allocated by **dce_aud_print()**, **dce_aud_get_header()**, or **dce_aud_get_ev_info()**.

Closing the Audit Trail File

Finally, the audit trail file from which the audit records were read must be closed using the **dce_aud_close()** function.

Chapter 36. The Password Management API

User passwords are the weakest link in the chain of DCE security. Users, unless their choices are restricted, typically choose passwords that are easy for them to remember; unfortunately, these memorable passwords are also easy for attackers to “crack.”

The password management facility is intended to reduce this risk by providing the tools necessary to develop customized password management servers, and to call them from client password change programs. This facility enables cell administrators to

- Enforce stricter constraints on users’ password choices than those in DCE standard policy
- Offer, or force, automatic generation of user passwords

The password management facility includes the following APIs:

- The password management interface, **sec_pwd_mgmt_ *()**, which enables clients to retrieve a principal’s password management ERA values and to request strength-checking and generation of passwords.
- The password management network interface, **rsec_pwd_mgmt_ *()**, which enables a password management server to accept and process password strength checking and generation requests.

Figure 84 provides a schematic view of the relationships and usages of these interfaces, as well as some relevant security registry APIs. This chapter first discusses the client API and then the network API.

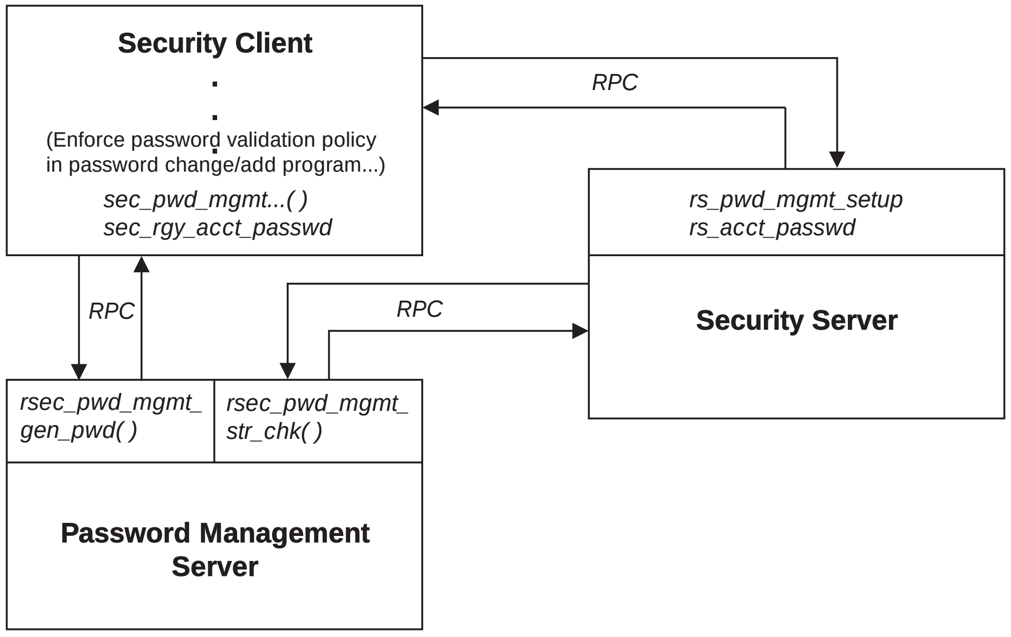


Figure 84. Use of Password Management Facility APIs

For information on how to administer password generation and strength-checking, see the *OSF DCE Administration Guide—Core Components*.

The Client-Side API

The DCE control program, **dcecp**, and **rgy_edit** provide support for password generation based on a principal's password validation type ERA. However, if you want to enhance your own password change program (such as the UNIX **passwd** program), you will need to use the client-side **sec_pwd_mgmt_ *()** API.

This API provides functions that retrieve a principal's password management ERA values and request password strength checking and generation from a password management server.

The **sec_pwd_mgmt_ *()** API is defined in the **sec_pwd_mgmt.idl** file.

The general procedure for using the client-side password management API in a password change program is as follows. Refer to Figure 84 on page 621 as you read the following steps:

1. The client calls **sec_pwd_mgmt_setup()**, specifying the login name of the principal whose password is being changed. The registry service returns the **pwd_val_type** and **pwd_mgmt_binding** ERAs as well as the registry standard (password) policy for the principal to the client's security runtime, which is stored in a password management handle (an opaque data type).
2. The client calls **sec_pwd_mgmt_get_val_type()**, specifying the handle returned by **sec_pwd_mgmt_setup()** in step 1. The value of the principal's **pwd_val_type** ERA is extracted from the handle and returned to the client.
3. The client analyzes the principal's **pwd_val_type** ERA to determine whether a generated password is required. If so, it calls **sec_pwd_mgmt_gen_pwd()**, specifying the number of passwords needed, and the handle returned by **sec_pwd_mgmt_setup**. The client security runtime makes an RPC call to the password management server, which generates passwords that adhere to the principal's password policy.
4. The client calls **sec_rgy_acct_passwd()** (or some other form), specifying the new password (either input by the user or generated by **sec_pwd_mgmt_gen_pwd()**). If the principal's **pwd_val_type** ERA mandates it, the registry service makes an RPC call to the password management server, specifying the name of the principal and the password to be strength checked. The password management server checks the format of the password according to the user's password policy and accepts or rejects it.
5. The client calls **sec_pwd_mgmt_free_handle()** to free the memory associated with the password management handle.

Following is an example of a password change program that calls the **sec_pwd_mgmt_ *()** API as previously described.

```
sec_pwd_mgmt_setup(&pwd_mgmt_h, context, login_name,
  login_context, NULL, &st);
if (GOOD_STATUS(&st)) {
  sec_pwd_mgmt_get_val_type(pwd_mgmt_h, &pwd_val_type, &st);
}
if (GOOD_STATUS(&st)) {
  switch (pwd_val_type) {
    case 0: /* NONE */
    case 1: /* USER_SELECT */
      ... get password ...
      break;
    case 2: /* USER_CAN_SELECT */
      ... if user does not want generated password ... {
        ... get password ...
        break;
      }
    case 3: /* GENERATION_REQUIRED */
      sec_pwd_mgmt_gen_pwd(pwd_mgmt_h, 1, &num_returned,
        &passwd, &st);
      ... display generated password to user - possibly
        prompting for confirmation ...
      break;
  }
}
if (GOOD_STATUS(&st)) {
  sec_rgy_acct_passwd(context, &login_name, &caller_key,
    &passwd, new_keytype, &new_key_version, &st);
}

sec_pwd_mgmt_free_handle(&pwd_mgmt_h, &st);
```

The Password Management Network Interface

The password management interface, **rsec_pwd_mgmt_ *()**, provides a DCE-common interface to password management servers. It is the interface exported by the sample password management server provided with DCE Version 1.1 (**pwd_strengthd**), and it is the interface that application developers should use to write their own password management servers. Developers should use the sample code provided as a base for enhancements.

The API is defined in the **rsec_pwd_mgmt.idl** file.

Implementations must conform to the **rsec_pwd_mgmt_gen_pwd(3sec)** and **rsec_pwd_mgmt_str_chk(3sec)** reference pages.

The **rsec_pwd_mgmt_ *()** routines are

- **rsec_pwd_mgmt_gen_pwd()**
  Generates one or more passwords for a given principal.
- **rsec_pwd_mgmt_str_chk()**
  Strength checks a principal’s password according to policy.

Chapter 37. The DCE Certification Service

The DCE certification service provides for the secure retrieval of public keys, stored (through the DCE directory service) under the names of the principals with which the keys are associated. It is a name-to-public key translation service intended to be used both by DCE components and DCE applications. The keys are stored in data structures called “certificates”.

Rules that define which entities are trusted to create certificates for which principals are embodied in policy modules, which have the job of retrieving, upon request, the public keys from the certificates (and verifying the certificates themselves when doing so).

DCE certification is a “secondary” facility, in that the service it provides is useful only in the context of some other application activity. Essentially, it does nothing but return public keys when presented with principal names (provided that the public keys have been properly stored under the names in the first place). It is then up to the application to do something useful with the keys.

This chapter is not intended to provide detailed guidance on how DCE applications should use public keys, although some discussion of public key usage is included. It is mainly concerned with explaining how DCE applications can use the certification service to store and retrieve the keys.

Who Needs to Use the Certification API?

The DCE certification service is intended to form one part of an implementation of a public key based authentication (and data protection) service in DCE. Thus the first-level users of the certification API will be various components of DCE itself; for example, RPC. However, the certification service can also be (and is intended to be) used by distributed applications that wish to use public keys for their own authentication or data protection purposes. The high-level public key retrieval routines are designed for this kind of use.

The low-level certification routines, on the other hand, are intended for applications that wish to implement and add new policies and/or cryptographic modules. For example, adding a new policy will involve the following development task(s):

- Implementing and registering a policy module (see below)

  For example, a mail application that wished to institute its own model for authenticating users by public key would need to have its own policy module.
- (Optional) Implementing and registering a cryptographic module (see below)

  Cryptographic modules implement the various signature algorithms required to allow policy modules to verify retrieved certificates. Policy modules are generally concerned only with signature verification, and (due to licensing constraints) signature generation functions are not supplied with the cryptographic modules provided with the DCE reference implementation. Applications that wish to use the public keys returned by the DCE certification facility will typically augment the supplied verification functions with signature generation routines.

Other possible users of the DCE certification API might be developers who wish to implement their own signature algorithms (cryptographic modules). (Signature algorithms are specified in a field in the certificate; they are selected at the time a certificate is created.) Only developers who wish to add to the available signature algorithms, or who wish to add signature generation capability to a supplied algorithm, will need to implement new cryptographic modules.

The low-level certification API is not intended to be accessed directly by run-of-the-mill DCE applications.

Overview of DCE Certification

In the discussion that follows, note that the term “principal” does not necessarily mean or imply “DCE principal”. In a general sense, a principal is any name that can be authenticated—that is, any name that has one or more associated key(s). A DCE principal (one that is registered in the DCE registry) has DES key(s) maintained within the registry, while a public key (PK) principal has one or more public keys (generally stored within certificates). The only situation in which a PK principal has to be a DCE principal is where an application is using the “registry retrieval” policy (see “Direct secd” Lookup: DCE Registry Lookup Policy Model’’ below), since this policy retrieves the principal’s public keys from a its registry entry.

The DCE certification service provides for the secure storage and retrieval (by principal name) of public keys. The keys are stored in the DCE directory service, under the principal names with which they are to be associated.

Principals’ public keys are thus easily accessible through the namespace. However, in order to be regarded as valid (certified), the public key information must be properly “signed” by the certifying authority (CA) authorized to deposit public key information for the principal in question. The public key, with the signature of the CA that issued it, is stored (together with various other data) in a format defined by the ISO 9594-8/X.509 standard and called a “certificate”. Just who the authorized certifying authority for a given certificate is is defined by the trust policy model applicable to the subject in whose name the certificate is issued.

The CA’s signature is in the form of a checksum on the public key encrypted with the CA’s own private key, and verifiable by decrypting with the CA’s public key. The certificates are thus secure from tampering by any entity but the authorized (according to the defined policy model) CA, which alone possesses the private key required to sign the data.

Use of Public Keys

The DCE certification service stores and retrieves “public” keys. The important characteristic of such keys is that they exist and operate as pairs. Messages encrypted under one of the keys can be decrypted by means of the other (and vice versa); but messages cannot be encrypted and decrypted by means of the same key.

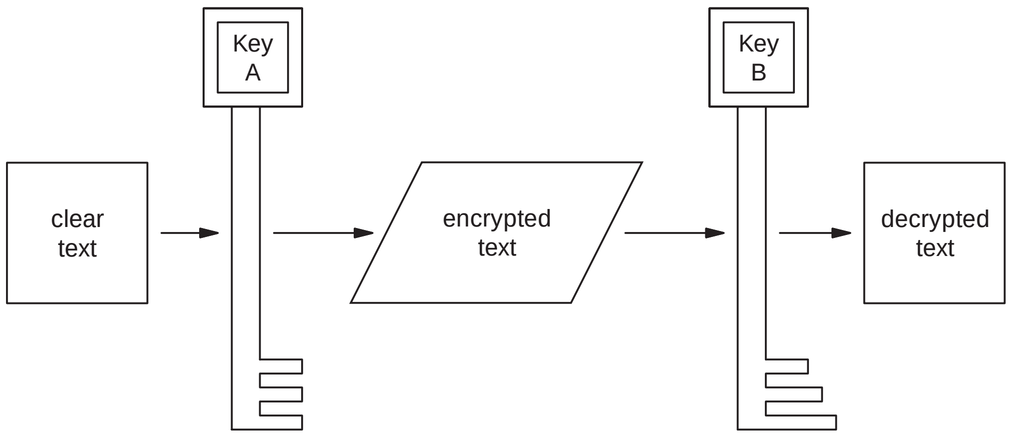


Figure 85. How Public Keys Work: Part 1

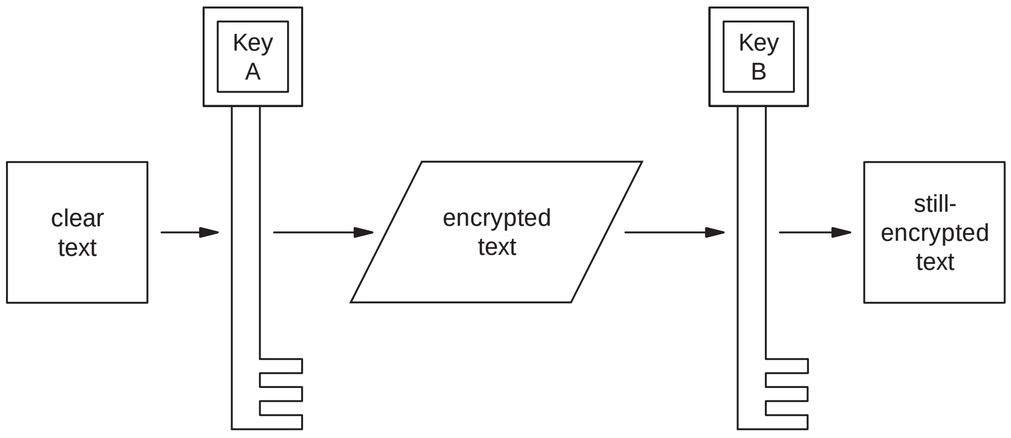


Figure 86. How Public Keys Work: Part 2

This asymmetric behavior of the public key-pair makes it ideal for network authentication purposes. One of the keys can be freely publicized in the network, and the other kept secret with, say, a server principal who desires to use it to authenticate itself. The server does this, whenever it is contacted by a prospective client, by simply encrypting a message under its secret key and sending it back to the client. The client then attempts to decrypt the message using the public key it knows belongs to the principal it wishes to contact. If it can decrypt the message, it regards the server as having authenticated itself. The same procedure can be used to authenticate the client (using a different key-pair).

An important detail in the above scenario is that the prospective client “knows” that the public key it uses to decrypt the server’s message really is that server’s public key. The other (unmentioned) detail is that the client has to get the public key from somewhere. The secure distribution of the public keys is the job of the DCE certification service.

It is not desirable to have all the public keys indiscriminately accessible to everybody, because then no one will have a reliable criterion for believing whose key is whose. The public keys must be deposited in such a way that users can always be sure that a given public key really “belongs” to the principal it is supposed to belong to; otherwise entities will be able to impersonate each other simply by switching public keys in the database they are retrieved from. Thus there needs to be some way to make sure that only authorized entities have access to principals' public keys.

The certification service "certifies" public keys by storing them with "signatures" generated by the distributors of the keys. The authenticating signatures on certificates are themselves implemented by public/private key pairs. A signature is simply the data of which the certificate consists, encrypted under the issuer's private key. A potential user of the certificate must possess the certificate issuer's public key. The issuer's public key can then be passed (along with the certificate contents, including the signature) to a library routine that will check whether the signature can be successfully decrypted to produce the information in the rest of the certificate. If the signature thus tested is found to be authentic, the user of the certificate can be certain that it was issued by the entity whose public key it checked against the certificate signature — namely, the principal that is supposed to have issued the certificate. The public key signature thus ensures the authenticity of data that can be distributed (and thus easily accessed) via the namespace.

A principal's public key can also be used by entities to protect data being sent to the principal. Data encrypted under the public key can be decrypted only by the possessor of the private key.

Contents of Certificates

The primary information that any certificate contains is the public key that is to be associated with some principal name. "Issuance" of a certificate means that the certificate is deposited into the name service, and attached (as a directory attribute) to the principal name it is to be associated with. Certificates are issued by certifying authorities (CAs); the CA's signature on the certificate is what certifies the public key information that the certificate contains.

A certificate contains the following information:

subject name
: The name of the principal for whom the certificate was issued. This is the name under which the certificate contents will be read by users.

issuer name
: The principal name of the issuer of the certificate, a CA (certifying authority) authorized to issue certificates for the subject.

version number
: Identifies the X.509 format version of the certificate.

serial number
: The certificate serial number, used to identify certificates in certificate revocation lists (CRLs).

start time
: The time from which the certificate's contents are considered to be valid.

end time
: The time until which the certificate's contents are valid.

signature algorithm
: An OID (object identifier) that identifies the algorithm used to encrypt the certificate signature.

parameters
: Any parameters necessary to pass to the signature verification algorithm.

signature
: A checksum of the certificate data, encrypted under the certificate issuer's private key, successful verification of which, by means of the issuer's public key, constitutes authentication of the certificate.

subject key
: The public key that is to be associated with the subject of the certificate (named by "subject name").

subject UUID
: (Optional) A UUID that identifies the certificate subject.

issuer UUID
: (Optional) A UUID that identifies the issuer of the certificate.

The most important ingredients of a certificate are: the principal name which it is stored under; the public key which it contains; and the signature of the CA that issued it. These can be illustrated as shown in the following diagram:

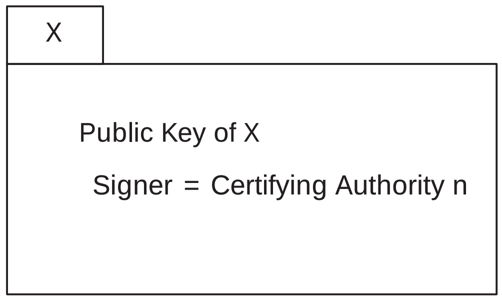


Figure 87. The Essential Parts of a Certificate

Component Parts of the DCE Certification API

The DCE certification API is organized into four groups of routines:

- Routines for implementing and registering cryptographic modules

  Cryptographic modules embody the signature algorithms that are used to sign and verify certificates. Certificates are signed by certifying authorities (which are usually invoked by system administrators or some other specially privileged authority to create certificates), and are retrieved (and verified) by policy modules (which are called by various applications seeking principals' public keys).
- Low-level certificate access and manipulation routines

  These routines represent the primitive certificate access operations which are used in the implementation of policy modules.
- Routines for implementing and registering policy modules

  Policy modules embody the rules and mechanisms for finding the public keys that are associated with some specific set of principals.

  High-level routines for use by applications that wish to access the certification service

The following diagram shows how these four groups of functionality are related to each other and to their two main groups of user: namely, system administrators and DCE applications.

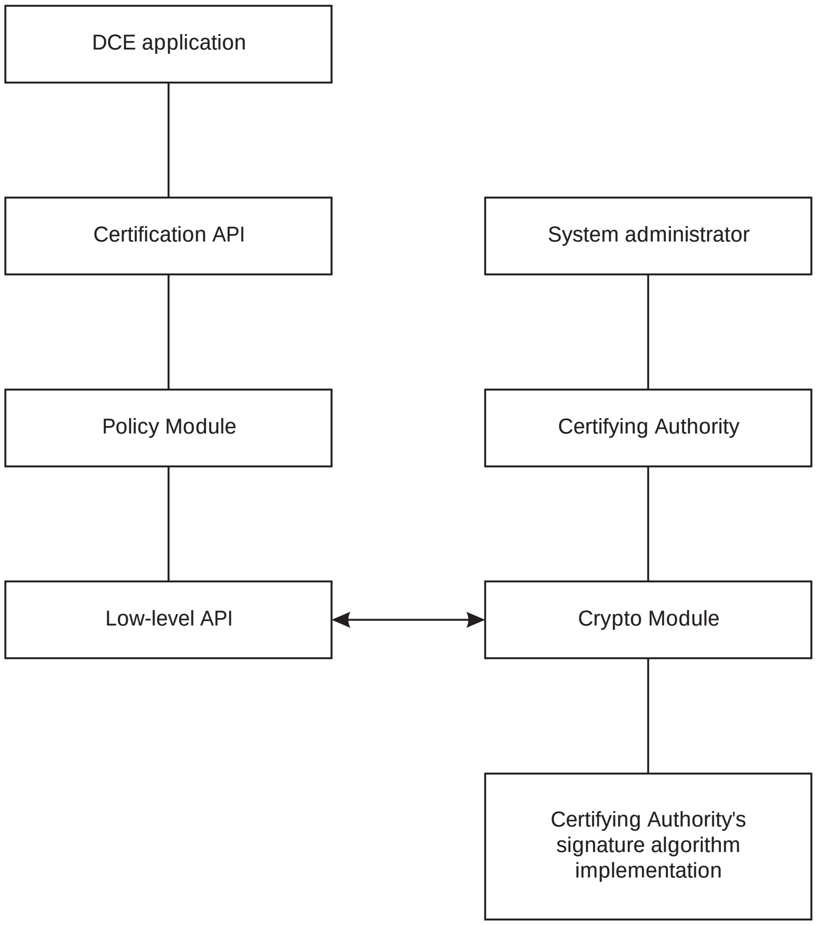


Figure 88. Certification API Organization

Note that certifying authorities merely create the certificates and deposit them in a place from which they can be retrieved (the namespace); they play no part in the retrieval process itself. In fact, this could be said to be the main reason for certificates in the first place: they allow a facility such as the directory service to be used as the distribution point for public keys (that is, they allow an application to not have to arrange for getting its keys to prospective clients by some private mechanism), and at the same time they assure users that the key information that they contain has not been tampered with.

High Level Certification API

The following certification API routines are intended for general DCE application use:

- **pkc_get_registered_policies(3sec)**
- **pkc_init_trustlist(3sec)**
- **pkc_append_to_trustlist(3sec)**
- **pkc_init_trustbase(3sec)**
- **pkc_retrieve_keyinfo(3sec)**
- **pkc_get_key_count(3sec)**
- **pkc_get_key_data(3sec)**
- **pkc_get_key_trust_info(3sec)**

- **pkc_get_key_certifier_count(3sec)**
- **pkc_get_key_certifier_info(3sec)**
- **pkc_free_trustlist(3sec)**
- **pkc_free_trustbase(3sec)**
- **pkc_free_keyinfo(3sec)**
- **pkc_free(3sec)**

Key retrieval consists basically of two operations:

1. Generating an “initial trust base”—a starting point for future certification paths, consisting of a list of principals and their keys. An application would normally generate its initial trust base on startup.
2. Using the trust base to retrieve key(s) for a specified principal.

In outline, a typical pattern for an application’s use of the high-level API might proceed according to the following series of calls:

1. **pkc_get_registered_policies(3sec)**

   Called once for the lifetime of the application. It returns a set of OIDs, which point to all currently installed policies.
2. **pkc_init_trustlist(3sec)**

   The caller creates an empty “trust list” to hold the set of certificates it initially trusts.
3. **pkc_append_to_trustlist(3sec)**

   Called one or more times, to add certificates or keys which the caller trusts to its list of trusted keys.

   Steps 2 and 3 together build up the initial trust list.
4. **pkc_init_trustbase(3sec)**

   Computes a trust base, given the initial trust list. The caller uses one of the OIDs returned in Step 1, together with the list of trust items constructed in Steps 2 and 3, to access a policy and initialize a “trust base” containing all the certificates initially trusted under the specified policy, given the initial list of trusted keys.
5. **pkc_retrieve_keylist(3sec)**

   Called one or more times, for each individual’s public key that needs to be looked up.
6. **pkc_free_trustlist(3sec)**

   Frees storage allocated for the trust list.
7. **pkc_free_trustbase(3sec)**

   Frees storage allocated for the trust base.

Policy Models

A policy model (or trust policy model) is simply some scheme or set of rules that dictates which certifying authorities are authorized to issue certificates for which principals. In other words, the policy model will prescribe whose signature is to be regarded as a valid certifier for any given principal’s certificates. The policy module which embodies these rules will use them in verifying the certificates it reads from the namespace.

Since the certificates themselves are stored and accessed through the DCE directory service (either GDS or CDS), one obvious policy model will be to organize the certifying authorities' reponsibilities according to the same hierarchies. However, models that employ other certifying hierarchies, or no hierarchy at all, are also possible.

Certification Paths

The mechanism that the certification service uses to embody more complex models is certification paths. A certification path is implemented by a sequence of certificates. Rather than immediately accessing a given principal's certificate to determine its public key, the user must access the beginning of a chain of certificates in order to get to the final certificate that contains the desired principal's public key. The intervening certificates consist of a series of public keys of CAs, each certified by the next CA in the chain.

The following diagram shows how a certificate chain might be used to find the public key of a principal, **X**:

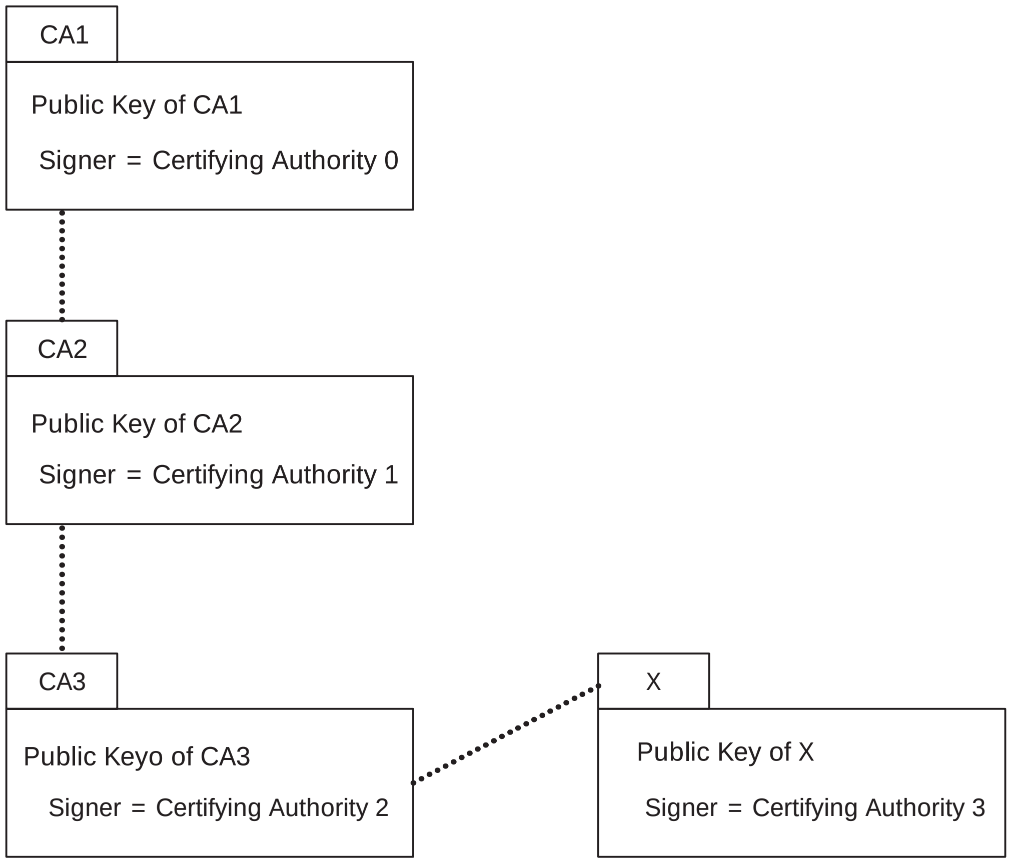

CA1
Public Key of CA1
Signer = Certifying Authority 0
CA2
Public Key of CA2
Signer = Certifying Authority 1
CA3
Public Keyo of CA3
Signer = Certifying Authority 2
X
Public Key of X
Signer = Certifying Authority 3

Figure 89. A Certificate Chain

In a policy model that uses certification paths, a given principal's public key is found by beginning with a certificate signed by a CA that is trusted by the entity requesting the public key (in the above diagram, the trusted CA is CA0). This certificate contains the public key of the next CA in the path, namely CA1. The policy module reads this certificate, learns the key of the next CA, CA2, and so on, until the certificate for X, the original target, is found.

The idea of certificate chains is to propagate authenticity via certifying authorities while not propagating the authorities' responsibility, thus reducing the effects of the compromise of single authorities, wherever they may exist in the hierarchy of authorities.

Implementing and Registering a Cryptographic Module

The routines in an application's cryptographic module make up the lowest level of functionality in the certification mechanism. Each module consists of a set of (at most) five routines, the most important of which are its **sign()** and **verify()** routines. (Note, however, that the **sign()** routine is not mandatory.)

Contents of a Cryptographic Module

Cryptographic modules are registered in the form of **pkc_signature_algorithm_t** structures, which contain the entry points for the following developer-supplied routines:

open()
Opens the module

close()
Closes the module

verify()
Verifies a certificate signature

sign() Affixes a signature to a certificate

verify() and **sign()** are the routines that will actually call the encryption/decryption functions appropriate to the algorithm.

name()
Returns the algorithm name, a character string that can be used in auditing or diagnostic messages.

The **pkc_signature_algorithm_t** structure also contains the following data fields:

- a version number

  Note that the version field of a cryptographic module is not the same thing as the version number of a certificate. A crypto module's version number is the version of the certification API that it is designed for (which in particular specifies the format of the **pkc_signature_algorithm_t** structure used to register the crypto module).
- an object identifier (OID) identifying the signature algorithm

Accessing a Registered Cryptographic Module

Signature algorithms are identified by object identifiers (the character string returned by **name()** is intended for use in diagnostic or auditing messages). Certificates contain a field which identifies by OID the algorithm used to sign that certificate.

Policy implementors are recommended to access cryptographic modules mainly through the following routines, which perform all locking necessary to make the calls thread safe, and also transparently handle any context information that a given cryptographic implementation may need.

- **pkc_crypto_get_registered_algorithms(3sec)**

Call this routine to get an OID set describing the currently registered algorithm implementations.

- **pkc_crypto_sign(3sec)**

  Call this routine to get data signed.
- **pkc_crypto_verify_signature(3sec)**

  Call this routine to verify signed data.
- **pkc_crypto_generate_keypair(3sec)**

  Call this routine to generate a pair of public/private keys.

Information about a cryptographic module may be obtained by calling **pkc_crypto_lookup_algorithm(3sec)**.

Data can also be signed and verified by looking up the desired algorithm (with **pkc_crypto_lookup_algorithm(3sec)**) and then explicitly calling the module's (sign)() or **verify()** routine, although in this case the calling application must take care to avoid multi-threading issues, and is also responsible for opening the crypto module prior to use, and closing it afterwards.

A list of the OIDs of all currently registered cryptographic modules can be obtained by calling **pkc_crypto_get_registered_algorithms()**. You can then access information about a specific module by calling the **pkc_crypto_lookup_algorithm()** routine. To sign data with a private key or to verify signed data with a public key, either **pkc_crypto_verify_signature()** or **pkc_crypto_sign()** can be called.

In the low-level certificate interrogation API, the **verify()** routine is automatically called by the **pkc_crypto_verify_signature(3sec)** routine.

Signature Algorithms Provided by DCE Certification

The signature algorithms provided with DCE 1.2.2 are **md2WithRSA** and **md5WithRSA**.

Registering a Cryptographic Module

Perform the following steps to register a cryptographic module:

1. Implement the **name()** and **verify()** functions. These two routines must be implemented.
2. If your module needs to perform any initialization or finalization tasks, implement **open()** and/or **close()** routines for them. (These two routines are optional.)
3. Implement **sign()** and **generate_keypair()** functions if necessary. (These two routines are optional.)
4. Create a **pkc_signature_algorithm_t** structure containing the entrypoints of the routines implemented in Steps 1 to 3 (use **NULL** for the entrypoint of any unimplemented routines), and use this structure to register the algorithm implementation.

Implementing and Registering a Policy Module

A certification trust model simply prescribes which certifying authorities (CAs) can legitimately issue (create and sign) certificates for which principals. A trust model is implemented by a policy module. The ultimate purpose of the certification service is to return public keys, and it is the job of the routines in a policy module to do this. Looked at from this point of view, policy modules are mainly distinguished from each other by the two following things:

- Which principals a policy module is willing to return keys for.
- Which CAs a module is willing to trust the signatures of on the certificates from which it retrieves keys.

A principal's certificates will be retrieved from a directory service entry (exactly what entry depends on the policy used), and the policy module will only look for certain signatures known to it on the certificates it retrieves. However, direct retrieval via the subject's directory entry name is only one trust model. There can be many others. For example, see the discussion of certification paths above.

Policy Modules Provided with DCE Certification

Several certification policy modules are provided by DCE. These policies are described in the following sections.

Direct secd Lookup: DCE Registry Lookup Policy Model

The registry lookup policy module simply looks up principals' public keys in the DCE registry, and returns them. These keys are not held in certificates, but are stored as extended registry attributes.

If a caller of the high-level certificate retreival API has DCE credentials, then the registry retrieval policy will authenticate the registry as part of the retrieval operation. If no credentials are available, no authentication is possible. In this case, keys will still be returned, but the certificate API will indicate to the caller that the keys are untrusted.

DCE Hierarchical Trust Policy

The DCE hierarchical trust policy supplied with DCE 1.2.2 supports hierarchical cells/DASS style trust paths. A trust path between principals A and B consists of zero or more up links, followed by zero or one cross link, followed by zero or more down links. The DCE hierarchical trust policy extension uses the DCE namespace certificate store extension (NCSE) for certificate retrieval.

Each cell is assumed to operate as a certification authority (CA) for top level principals registered within that cell. Thus, the CA for the cell **over_cell** is assumed to create a certificate for, say, the principal **felix** within **over_cell**, where the certificate signatory is "over_cell" and the certificate target is "over_cell/felix". If a cell employs structured names for principals, each level is considered to act as a CA for its subordinate. For example, if cell "over_cell" contains a principal **admin/JohnSmith**, then **over_cell/admin/JohnSmith** is certified by two certificates, the first signed by **over_cell** and certifying **over_cell/admin**, and the second signed by **over_cell/admin** and certifying **over_cell/admin/JohnSmith.**

To avoid requiring that the cell's root CDS directory be used for storing certificates for the cell's principals, the DCE NCSE provided in DCE 1.2.2 allows any directory (in CDS or GDS) to be given an attribute (with OID **1.3.24.9.15**) that names a subdirectory (of the directory to which the attribute is given) within which certificates are to be stored. The value of this attribute is a string which will be appended to the name of the directory to give the name of a new directory within which certificates will be stored.

For example, if the root directory in the cell **over_cell** is the directory to which this “certificate directory” attribute is attached, and the attribute contains the value **principals**, the DCE NCSE will attempt to retrieve the certificate for **over_cell/P** from the CDS directory **over_cell/principals/P**. If the “certificate directory” attribute is missing or empty, the cell root directory will be searched for principal certificates. (Note that the insertion of the “certificate directory” attribute value applies only to locating certificates within the directory service; the above certificate would contain the name **over_cell/P** as its actual subject.)

At most one “certificate directory” attribute is considered when looking for certificates for a given name, according to the following algorithm:

- Starting with the name for which certificates are desired, RDNs are removed from the right of the name until either a directory is found that contains the “certificate directory” attribute, or the namespace root is reached.

In general, CDS administrators should define the “certificate directory” attribute within each CDS root directory, rather than storing certificates within the root directory. As well as reducing clutter in the cell root directory, doing this has efficiency benefits (terminating the search for the certificate directory at each CDS root), and also prevents the definition of this attribute at a higher level within the DCE global namespace from influencing the placement of certificates within a cell’s namespace.

PEM-like Policy

No explicit PEM-like policy is provided with DCE 1.2.2; however, the DCE hierarchical policy may be used in a PEM-like fashion by specifying root CA keys as the initial trust list, rather than keys belonging to the caller’s immediate CA.

The Low Level Certificate Manipulation API

The certificate manipulation API is a C++ interface. C++ must be used to retrieve the certificates into trust lists and manilupulate them there.

The contents of the

```
/usr/include/dce/asn.h
```

and

```
/usr/include/dce/x509.h
```

header files define some of the basic types used by the low-level certificate manipulation routines, including the actual structure of certificates. Following is a list of the low-level certificate routines defined in the

```
/usr/include/dce/pkc_certs.h
```

file:

- **pkc_add_trusted_key(3sec)**
- **pkc_lookup_keys_in_trustlist(3sec)**
- **pkc_lookup_key_in_trustlist(3sec)**
- **pkc_lookup_element_in_trustlist(3sec)**
- **pkc_check_cert_against_trustlist(3sec)**
- **pkc_revoke_certificate(3sec)**
- **pkc_revoke_certificates(3sec)**
- **pkc_delete_trustlist(3sec)**
- **pkc_copy_trustlist(3sec)**
- **pkc_display_trustlist(3sec)**

Policy Module Implementation

Implementation of a policy module consists essentially of writing **establish_trustbase()**, **delete_trustbase()**, **retrieve_keyinfo()** and **delete_keyinfo()** routines, and associated interrogation routines. The module will find certificates for principal names according to the rules set out for that module, verify their signatures, and return the public keys found in them to the original callers.

Certificate Revocation Lists (CRLs)

Certificate revocation lists are lists of certificates whose contents are no longer to be believed. Use of CRLs is policy-specific. **pkc_certs** provides objects for parsing and manipulating CRLs, and for using them to invalidate portions of a trust list.

Accessing a Registered Policy Module

Policy modules are registered in the form of **pkc_policy_t** structures, which contain the entry points for the following developer-written routines:

open()
: opens the module

close()
: closes the module

retrieve_keyinfo()
: returns the public key for a specified principal name

name()
: Returns the name of the policy.

establish_trustbase()
: Creates a trust base.

delete_trustbase()
: Deletes a trust base.

delete_keyinfo()
: Deletes a **keyinfo** handle.

get_key_count()
: Returns the number of keys a **keyinfo** handle contains.

get_key_data()
: Retrieves an individual key.

get_key_trust()
Returns the type of trust established for a specific key.

get_key_certifier_count()
Returns the number of certifiers in the trust path that certified a key.

get_key_certifier_info()
Returns information about a specific certifier of a key.

The **pkc_policy_t** structure also contains the following data fields:

- a certificate version number
- an object identifier (OID) identifying the policy module

Policy modules, similarly to signature algorithms (cryptographic modules), are identified by object identifiers (the character string returned by **name()** is intended for use in diagnostic or auditing messages).

Also similarly to cryptographic modules, there are two ways in which cryptographic modules can be accessed: either by a single call to which the identifying OID is passed (this is the recommended method); or by calling **pkc_plcy_lookup_policy(3sec)** and then (for example) the module's (***retrieve_key()**) routine to obtain the public key (a list of the OIDs of all currently registered policy modules can be obtained by calling **pkc_plcy_get_registered_policies()**).

Registering a Policy Module

You must implement the following routines in any policy module:

name()
Returns the name of the policy.

establish_trustbase()
Creates a trust base, which is a policy-specific data structure based on the initial set of trusted keys.

retrieve_keyinfo()
Given a trust base, returns a handle to keys for a specific principal.

delete_trustbase()
Deletes a trust base.

delete_keyinfo()
Deletes a **keyinfo** handle.

get_key_count()
Given a **keyinfo** handle, returns the number of keys it contains.

get_key_data()
Retrieves an individual key from a **keyinfo** handle

get_key_trust()
Returns the type of trust established for a specific key, and the purpose(s) for which that trust applies.

The following policy routines are optional:

open(), close()
These routines perform any initialization and/or finalization tasks required by the module.

get_key_certifier_count()
This routine is required only for policies that return **CERTIFIED_TRUST** keys; it returns the number of certifiers in the trust path that certified a key.

get_key_certifier_info()
This routine is required if the module implements **get_key_certifier_count()**. It returns information about a specific certifier with the certification path of a specific key. Certifier 0 is the immediate certifier of the key; certifier 1 is the CA that certified certifier 0, and so on.

Once you have implemented all necessary routines for you module, you must create a **pkc_policy_t** structure containing their entrypoints. Unimplemented routines' entrypoints should be specified as **NULL**.

Registering the module

The module is registered by calling the registration function and passing it a **pkc_policy_t** structure, which contains the entry points for the module routines described above:

pkc_plcy_register_policy()

Part 6. Appendixes

Index