Gradient® DCE for Tru64™ UNIX®

Product Guide

Software Version 4.2

Notices

Gradient DCE for Tru64 UNIX Product Guide - Software Version 4.2 - Revised November 2001

Entegrity Solutions Corporation, 2077 Gateway Place, Suite 200, San Jose, CA 95110, USA

Contents

Chapter 10 RPC, IDL, ACF, and IDL Compiler Enhancements 105

Chapter 11 Application Debugging with the RPC Event Logger 117

Chapter 12 Developing Distributed Applications with FORTRAN 137

Preface

Intended Audience

The audience for this guide includes the following:

- *Experienced programmers who want to write client/server applications.*
- *Experienced programmers who want to port existing applications to DCE.*
- *System managers who manage the distributed computing environment.*
- *Users who want to run distributed applications.*

Overview of this Guide

The *Gradient® DCE for Tru64™ UNIX® Product Guide* provides users of the Distributed Computing Environment (DCE) with supplemental information necessary to use Gradient DCE. This guide is best used with the documents listed under Related Documentation .

Gradient DCE for Tru64 UNIX v4.0 is a layered product on the Tru64 UNIX Version 5.0, 5.0*a*, and 5.1operating systems. It is a compatible upgrade of DCE for Tru64 UNIX Version 3.0.. It consists of a full DCE implementation as defined by The Open Group (TOG). This software includes these components:

- *Remote Procedure Call (RPC)*
- *Cell Directory Service (CDS)*
- *Distributed Time Service (DTS)*
- *DCE Security*
- *DCE Distributed File Service (DFS, based on DCE Release 1.2.2)*
- *Lightweight Directory Access Protocol (LDAP)*

NOTE: The products named Gradient DCE for Tru64 UNIX v3.1 (and higher), Digital DCE v3.1, and Compaq DCE v3.1 provide essentially the same features; however, only Gradient DCE for Tru64 UNIX functions on the Tru64 UNIX v5.x operating system. Although other company names may be referenced within this document (Digital, Compaq, or Gradient Technologies), this DCE product is now produced and supported by Entegrity Solutions® Corporation.

Conventions

The following conventions are used in this guide:

UPPERCASE and lowercase	The operating system differentiates between lowercase and uppercase characters. Literal strings that appear in text, examples, syntax descriptions, and function definitions must be typed exactly as shown.
bold	Boldface type in interactive examples indicates typed user input. In general text reference, bold indicates file names and commands.
italics	Italic type indicates variable values, placeholders, and function argument names.
`special type`	Indicates system output in interactive and code examples.
`%`	The default user prompt is your system name followed by a right angle bracket (>). In this manual, a percent sign (%) is used to represent this prompt.
#	A number sign (#) represents the superuser prompt.
Ctrl/*x*	This symbol indicates that you hold down the Ctrl key while pressing the key or mouse button that follows the slash.
<Return>	Refers to the key on your terminal or workstation that is labeled with Return or Enter.

Related Documentation

The following documents are available in HTML and Acrobat format on the Entegrity software CD:

- *Gradient DCE for Tru64 UNIX Installation and Configuration Guide* — Describes how to install DCE and configure and manage your DCE cell.
- *Gradient DCE for Tru64 UNIX Product Guide* (this guide) — Provides supplemental documentation for Gradient DCE for Tru64 UNIX value-added features.
- *Gradient DCE for Tru64 UNIX Reference Guide* — Provides supplemental reference information for Gradient DCE for Tru64 UNIX value-added features.
- *Gradient DCE for Tru64 UNIX Release Notes*— Lists new features, bug fixes, and known problems and restrictions.

The following OSF DCE Release 1.2.2 technical documentation is provided on the Entegrity software CD in PDF format:

- *Introduction to OSF DCE* — Contains a high-level overview of DCE technology including its architecture, components, and potential use.
- *OSF DCE Administration Guide - Introduction* — Describes the issues and conventions concerning DCE as a whole system and provides guidance for planning and configuring a DCE system.
- *OSF DCE Administration Guide - Core Components* — Provides specific instructions on how core components should be installed and configured.

- *OSF DCE Application Development Guide - Introduction and Style Guide* — Serves as a starting point for application developers to learn how to develop DCE applications.
- *OSF DCE Application Development Guide - Core Components* — Provides information on how to develop DCE applications using core DCE components such as RPC and security.
- *OSF DCE Application Development Guide - Directory Services* — Contains information for developers building applications that use DCE Directory Services.
- *OSF DCE Application Development Reference* — Provides reference information for DCE application programming interfaces.
- *OSF DCE Command Reference* — Describes commands available to system administrators.

Contacting Entegrity Solutions

Contact	Address	Phone/Fax/E-mail
DCE Product and Sales Information	Entegrity Solutions Corporation 410 Amherst Street, Suite 150 Nashua, NH 03063 USA	E-mail: DCESales@entegrity.com Web: **www.entegrity.com** Telephone and Fax: United States and Canada Tel: +1 (603) 882-1306 Tel (US Only): 1-800-525-4343 Fax: +1 (603) 882-6092
All Other Product and Sales Information Requests	Entegrity Solutions Corporation 2077 Gateway Place, Suite 200 San Jose, CA 95110 USA	E-mail: info@entegrity.com Web: **www.entegrity.com** Telephone and Fax: Tel: +1 (408) 487-8600 Tel (US Only): 1-866-487-8600 Fax: +1 (408) 487-8610
Technical Support	Entegrity Solutions Corporation Technical Support 2 Mount Royal Ave. Marlborough, MA 01752 USA	United States and Canada: Tel: +1 (508) 229-0239 Tel (US Only): 1 (888) 368-3555 Fax: +1 (508) 229-0338 E-mail: support@entegrity.com **http://support.entegrity.com**

Obtaining Technical Support

If you purchased your NetCrusader™ product directly from Entegrity Solutions Corporation or Gradient Technologies, Inc. you are entitled to 30 days of limited technical support beginning on the day the product is expected to arrive.

You may also purchase a support plan that entitles you to additional services. You *must* register prior to receiving this support. For details, refer to the customer support information package that accompanied your shipment or refer to the Technical Support area of **http://support.entegrity.com**. The web site also contains online forms for easy registration.

If you purchased NetCrusader from a reseller, please contact the reseller for information on obtaining technical support.

Obtaining Additional Technical Information

Contact	Address	Phone/Fax/Email
The Open Group™ Developer of DCE (Distributed Computing Architecture) software and standards.	The Open Group™ 29B Montvale Ave Woburn MA 01801 U. S. A.	Tel: +1 781-376-8200 Fax: +1 781-376-9358 **http://www.opengroup.org**

Obtaining Additional Documentation

All documentation for your NetCrusader product is provided in electronic format on the same CD on which the product ships. See the product CD for information on accessing this documentation.

Documentation for all of Entegrity's products is available at **http://support.entegrity.com**. Enter the Support Web area and click the Documentation link.

We are always trying to improve our documentation. If you notice any inaccuracies or cannot find information, please send email to **docs@entegrity.com**. We welcome any comments or suggestions.

CHAPTER 1

Gradient DCE for Tru64 UNIX

1.1 Overview of the Software

Distributed computing services, as implemented in the Distributed Computing Environment (DCE), provide an important enabling software technology for the development of distributed applications. DCE makes the underlying network architecture transparent to application developers. It consists of a software layer between the operating system and network interface and the distributed application. It provides a variety of common services needed to develop distributed applications, such as name, time, and security services, and a standard remote procedure call interface.

Gradient® DCE for Tru64™ UNIX® provides a means for application developers to design, develop, and deploy distributed applications. Gradient DCE for Tru64 UNIX is based upon OSF® DCE Release 1.2.2.

1.2 Kit Contents

Gradient DCE for Tru64 UNIX consists of the following distributed computing technologies:

- DCE Remote Procedure Call (RPC), which allows you to create and run client/server applications.
- DCE Cell Directory Service (CDS), which provides location-independent naming for servers.
- DCE Distributed Time Service (DTS), which synchronizes time in distributed network environments.
- DCE Security Service, which provides secure communications and controlled access to resources.
- DCE Distributed File Service (DFS), which provides transparent file access to a single namespace in a distributed computing environment.

The Gradient DCE for Tru64 UNIX product consists of twelve subsets:

- Runtime Services (RTS) subset
- Security Server (SEC) subset
- Cell Directory Server (CDS) subset
- Six Distributed File Service (DFS) subsets
- Application Developer’s Kit (ADK) subset
- Online Command Reference Manual Pages (MAN) subset

- Online Application Developer's Manual Pages (ADKMAN) subset

The rest of this chapter describes the subsets, additional support, and restrictions for this product.

1.2.1 Runtime Services (RTS) Subset

You must install the DCE Runtime Services subset on all systems on which you want to run DCE applications. This subset includes the DCE client software necessary to run DCE distributed applications and the administrative tools required to configure and maintain the DCE environment. This subset is a prerequisite for all the other subsets. For DCE server configurations, you must install the appropriate subsets described in the following sections. .

Specifically, this subset includes the following components:

- Remote Procedure Call runtime library

  The RPC runtime library includes routines that manage communications between client and server stubs. The DCE host daemon (**dced**) maintains the endpoint map (addresses). The RPC Event Logger (**rpclm**) is also provided in this subset.

- Cell Directory Server (CDS) clerk and CDS advertiser

  The CDS clerk (**cdsclerk**) runs on the client node and serves as an intermediary between client applications and CDS servers. Clerks learn about CDS servers by listening to messages sent out by the CDS advertiser (**cdsadv**).

- DIGITAL X.500 XDS Library

  The XDS library provides support for use of XDS with DCE naming.

- Distributed Time Service (DTS) clerk and server

  The DTS clerk and server (**dtsd**) synchronize time in distributed network environments.

- PC name service interface daemon

  The Runtime Services subset provides the name service interface daemon (**nsid**), also called the PC Nameserver Proxy Agent, to allow interoperability with machines running Microsoft® RPC on MS-DOS®, Windows®, Windows NT™, or Windows 95™. If the PC is running DCE services, the **nsid** is not necessary.

- Audit Service daemon

  The Audit Service daemon (**auditd**) records and logs significant events (such as creating a user, logging in, or obtaining a ticket) in an audit trail file. Application servers can be designed to use the Audit Service for logging purposes.

- Administrative tools

The administrative tools include the control program **dcecp** and the enhanced CDS Browser (**cdsbrowser**). The Browser provides a graphical user interface for viewing the CDS namespace. The **dcecp** program replaces the earlier control programs **rpccp** for RPC, **cdscp** for CDS, and **dtscp** for DTS. These control programs remain available for special purposes, however.

The administrative tools also include **rgy_edit**, **passwd_import**, **passwd_export**, **acl_edit**, **getcellname** and **sec_admin**.

- DCE Configuration Program

  The DCE configuration program (**dcesetup**) allows you to configure your DCE environment.

- DCE Login Facility

  The **dce_login** command allows you to log in to DCE.

- Other DCE tools

  kinit
  kdestroy
  klist
  UUID Generator (**uuidgen**)

1.2.2 Cell Directory Server Subset

The Cell Directory Service (CDS) Server subset provides a consistent mechanism for naming and locating users, applications, files, and systems within a DCE cell. The CDS Server subset requires the Security Server subset to be installed. The CDS Server subset includes the following components:

- CDS server (**cdsd**)
- Global Directory Agent (**gdad**)

The Global Directory Agent (GDA) allows you to link multiple CDS or LDAP namespaces using the Internet Domain Name System (DNS) or X.500. To link multiple CDS namespaces using X.500, you must install the DIGITAL X.500 Base kit and the DIGITAL X.500 API kit on your CDS server. Optionally, you can install the DIGITAL X.500 Administration Facility kit for debugging and general administrative support. LDAP directory services using X.500 can be enabled during configuration.

The XDS library allows applications to access the CDS and X.500 directory services. The XDS routine library reference pages are provided in the *Gradient DCE for Tru64 UNIX Reference Guide.*

1.2.3 Security Server Subset

The Security Server (SEC) subset provides secure communications and controlled access to resources in a DCE environment. DCE Security includes authentication, secure communication, and authorization. The Security Server subset includes these components:

- Security server (**secd**)

- Tool used to create the security database (**sec_create_db**)
- Tool used to move the security database (**sec_salvage_db**)

1.2.4 Application Developer's Kit Subset

The Application Developer's Kit (ADK) subset provides the files necessary to develop DCE client and server applications using RPC, CDS, DTS, and Security application programming interfaces. Specifically, this subset includes these components:

- Interface Definition Language (IDL) stub compiler
- Required DCE application development include files
- Sample time-provider routines
- Sample DCE applications
- Symbols and message strings (SAMS) compiler for building DCE message files, as described on the **sams** reference page.

1.2.5 Online Manual Pages Subset

Product Name provides two sets of online reference (manual) pages: administrative commands for managing DCE, and application development routines for programming distributed applications. To use the online reference pages on Tru64 UNIX systems, specify the command or routine name with the **man** command. For example, this command displays the reference page for **uuidgen**:

```
% man uuidgen
```

If more than one reference page exists for a topic (for example, **intro**), you must specify the section number. For example, this command displays the introduction reference page for security:

```
% man 3sec intro
```

For multiple-word commands, use underscore characters to connect the words in the command. For example, this command displays the reference page for **rpccp show entry**:

```
% man rpccp_show_entry
```

1.2.6 Distributed File Service Runtime Services Subset

You must install the Distributed File Service (DFS) Runtime Services subset on all systems on which you want to run DCE DFS. This subset provides runtime services, including the DCE DFS client software necessary to run DCE DFS, and the administrative tools required to configure and maintain the DCE DFS environment.

Specifically, this subset includes the following components:

- User-level commands that include the DCE DFS configuration program (**dfssetup**), Basic OverSeer (BOS) Server commands (**bos**, **bosserver**), Cache Manager commands (**cm**), user-space services to the Cache Manager or File Exporter (**dfsbind**), Cache Manager daemon (**dfsd**), non-DCE LFS partitions exporter (**dfsexport**), Fileset Location Database

Server daemon (**flserver**), Fileset Server and Fileset Location Server commands (**fts**), Fileset Server daemon (**ftserver**), File Exporter daemon (**fxd**), Update Server daemon for clients (**upclient**), and Update Server daemon for servers (**upserver**).

- Error message files
- DFS shared library (**libcedfs.so**)

1.2.7 DFS Kernel Binary Subset

This subset contains the kernel binary files.

1.2.8 DFS Utilities Subset

This subset contains DCE DFS utilities:

- **scout** displays File Exporter statistics.
- **dfstrace** helps you diagnose problems in the kernel or within server processes that interface with **dfstrace**.
- **udebug** displays Ubik status information.

1.2.9 DFS Online Manual Pages

This subset contains the online reference (manual) pages for the administrative commands for managing DCE DFS. See *Section 1.2.5 on page 18* for information on displaying the online manual pages.

1.2.10 NFS-DFS Secure Gateway Server

This subset contains the NFS-DFS Secure Gateway server components. It includes the gateway authentication daemon and the local authentication registration utility.

See the *Gradient DFS for Tru64 UNIX Configuration Guide* for more information on configuring DFS.

1.3 Platforms and Networks Supported by Gradient DCE for Tru64 UNIX

1.3.1 Interoperating with PCs

Your DCE server can interoperate with a PC client that has Microsoft RPC software installed on it. To use RPC from a PC, you need not make any changes to the DCE server system.

1.3.2 Network Support

Gradient DCE for Tru64 UNIX supports Tru64 UNIX Version 5.0a. Gradient DCE for Tru64 UNIX provides RPC communications over UDP/IP, TCP/IP, and the DECnet/OSI (Phase V) network protocol family. This network family includes both NSP and OSI transport protocols.

DECnet/OSI support provides upward compatibility for applications on OpenVMS™ nodes running DECnet™ Phase IV. Client applications that run over DECnet Phase IV on OpenVMS can use DECnet Phase IV addressing semantics to communicate with server applications running over the DECnet/OSI Phase V protocol families on Tru64 UNIX.

Application writers can establish a relationship between a client and server by specifying a protocol sequence in one of three ways: in an explicit string binding, in an interface definition, or by registering in the Cell Directory Service. (See the *OSF DCE Application Development Reference*, **intro(3rpc)**, for a list of valid DCE protocol sequences.) A server that prints addresses as part of its runtime operation prints network addresses and endpoint semantics as string bindings. (For a complete description of the format of string representations of binding information, see the *OSF DCE Application Development Guide*)

The following string bindings are examples for each protocol supported on Tru64 UNIX platforms. In any DCE implementation that supports the OSI protocol stack, whether from Entegrity Solutions® or another vendor, string bindings such as these are printed when an application fully implements all supported protocols.

String Bindings for the IP protocol:

```
ncacn_ip_tcp:16.20.16.155[3924]
ncadg_ip_udp:16.20.16.155[1575]
```

String Bindings for the DECnet Phase IV protocol:

```
ncacn_dnet_nsp:12.36[RPC2DD20001]
```

String Bindings for the DECnet/OSI (Phase V) protocols:

```
ncacn_osi_dna:%x49000caa000400243021[RPC52DD20001,tpid=cots]
ncacn_osi_dna:%x49000caa000400243020[RPC52DD20001,tpid=nsp]
ncacn_osi_dna:NODENAME[RPC52DD20001,tpid=cots]
```

1.4 Threads

The Pthreads interface is an important part of the architecture for DCE, and the DCE services rely on it. DCE uses the Pthreads interface from POSIX 1003.4a/d4. DECthreads is provided as part of the Tru64 UNIX operating system. Refer to the *Guide to DEC Threads* in the operating system's documentation set for information about threads.

1.5 Enhancements to OSF DCE

The Gradient DCE for Tru64 UNIX kit provides the following added-value features, which are not included in the OSF offering, to help users develop and deploy DCE applications:

- CDS Enhanced Browser
- IDL compiler enhancements
- RPC Event Logger Utility
- NSI daemon (PC Nameserver Proxy Agent)
- Security Integration Architecture (SIA)
- RPC support of DECnet/OSI (Phase V)
- DTS support of DECnet/OSI (Phase V)
- CDS Cache Clerk Enhanced Memory Management
- CDS Preferencing
- DTS Support for RPC and DLI (Data Link Interface)
- LDAP Directory Service
- New localrpc protocol sequence
- Kerberos 5-compliant utilities
- DCE in a Tru64 UNIX TruCluster Application Server Environment

1.5.1 CDS Enhanced Browser

The CDS Enhanced Browser contains additional functions beyond those contained in the OSF DCE Version 1.1 Browser. See *Chapter 5* for more information.

1.5.2 IDL Compiler Enhancements

The Gradient DCE for Tru64 UNIX IDL compiler in this kit includes important enhancements, which are Entegrity value-added functionality available only with Gradient DCE for Tru64 UNIX:

- Runtime routine templates
- DEC Fortran support

See *Chapter 10* for more information about IDL.

1.5.3 The RPC Event Logger Utility

Entegrity provides the RPC Event Logger, which records information about operations relating to the execution of an application interface.

1.5.4 Name Service Interface Daemon for Microsoft RPC

Entegrity provides the name service interface daemon (**nsid**), also known as the PC Nameserver Proxy Agent, to allow RPC communication with personal computers running the DCE-compatible Microsoft RPC. The **nsid** enables an RPC application on MS-DOS, DOS Windows, and Windows NT to perform name-service operations that are available through RPC, as if the RPC applications on the PC were directly involved in the full CDS namespace.

1.5.5 Security Integration Architecture

Security Integration Architecture (SIA) lets users of Gradient DCE for Tru64 UNIX use both BSD security and DCE security by using the same system commands and routines for both.

1.5.6 RPC Support of DECnet/OSI (Phase V)

This version of Gradient DCE for Tru64 UNIX supports Entegrity's DECnet/OSI implementation. See *Section 1.3 on page 19* for more information.

1.5.7 DTS Support of DECnet/OSI (Phase V)

This version of the Gradient DCE for Tru64 UNIX supports full functionality of DECnet/OSI implementation.

1.5.8 CDS Cache Clerk Enhanced Memory Management

The CDS enhanced command, **dcecp cdscache discard**, lets an administrator release specified structures from the cache without any need to stop and restart DCE.

1.5.9 CDS Preferencing

This enhancement improves performance at CDS clients by providing a ranking to the order in which clearinghouses are contacted by the client for CDS information. This can be accomplished automatically through the use of defaults associated with the location of CDS clients with respect to CDS servers or by manual overrides made by cell administrators. For more information, see *Section 5.4 on page 69*.

1.5.10 DTS Support for DLI (Data Link Interface) and RPC

This version of the Gradient DCE for Tru64 UNIX allows the acceptance of messages on both RPC (a new default) and DLI (the old default).

1.5.11 LDAP Directory Service

The Lightweight Directory Access Protocol (LDAP) provides access to the X.500 directory service without the overhead of the full Directory Access Protocol (DAP). LDAP supports the TCP/IP protocol.

1.5.12 New localrpc Protocol Sequence

Gradient DCE for Tru64 UNIX now supports a new protocol sequence. It is implemented with UNIX Domain sockets and can only be used by clients and servers that are on the same node. By using UNIX Domain sockets, the IP layer can be bypassed, providing gains in performance that may vary with the nature of the RPC traffic. The user must explicitly choose to use the *localrpc* protocol sequence in either a well-known endpoint in the IDL file, or as called

out by one of the rpc_server_use_protseq*() functions wherever a protocol sequence string can be used. String bindings can also be used to pass *localrpc* binding information from server to client.

1.5.13 Kerberos 5-Compliant Utilities

Massachusetts Institute of Technology (MIT) Kerberos Version 5 authentication and key distribution service is supported. The Kerberized secure and encrypting versions of UNIX network utilities are supported: telnet, rlogin, rsh, and ftp.

1.5.14 DCE in a Tru64 UNIX TruCluster Application Server Environment

Compaq TruCluster™ Solutions is a fault-resilient technology that maximizes uptime for mission-critical applications, databases, and operating systems. Gradient DCE for Tru64 UNIX is Compaq TruCluster tolerant.

1.6 Diskless Support Removed from OSF DCE

Support for diskless workstations was removed from OSF DCE Release 1.1. Consequently, Gradient DCE for Tru64 UNIX does not support diskless workstations.

1.7 Restrictions Using Gradient DCE for Tru64 UNIX

This section describes the following restrictions:

- Use of DCE DFS
- Use of the **chpass** utility
- Use of some features of the XDS Directory Interface

1.7.1 DCE DFS Restrictions and Limitations

For this release, Gradient DFS for Tru64 UNIX is based on OSF DCE Release 1.2.2, and has the following restrictions:

- Supports the UNIX® File System (UFS) and POLYCENTER™ Advanced File System (AdvFS) only. Enhanced DFS server capabilities (for example, fileset cloning) are not supported on the server side.
- DFS built-in backup is not supported. Instead, use the Tru64 UNIX native file system backup facility.
- The NFS-DFS Secure Gateway server does not support remote DFS login / logout capabilities. For authenticated access to DFS, users of DCE-unaware NFS clients must authenticate to DCE from the Gateway Server machine using a **dfsgw add** operation. Refer to the *OSF DCE DFS Administration Guide and Reference* for information about authenticating from a Gateway Server machine.

- Gradient DFS for Tru64 UNIX uses Tru64 UNIX ACLs. These lack some of the features of DCE ACLs. *Chapter 8* discusses how Gradient DCE for Tru64 UNIX compensates for some ACL limitations.
- DCE DFS uses DCE credentials to authorize access to objects in the DCE DFS namespace. Background daemons lacking DCE Security credentials may not be able to use the DCE DFS namespace. Processes started by system daemons (that is, **cron(8)**, **inetd(8)**, **rdistd(8)**) may be denied access to files in the DCE DFS namespace if they do not have DCE credentials. For example, attempts to use the commands **at(1)** or **rdist(1)** when the remote files are in the DCE DFS namespace may fail. This is an important security feature of DCE and DFS. See *Section 8.5 on page 100* for information on obtaining DCE credentials by using the **-k** flag with **dce_login**.

1.7.2 Utility Restriction

If SIA is enabled, the registry information change commands, **passwd** (change password), **chsh** (change shell) and **chfn** (change finger information) can alter the information in one of the configured security mechanism registries.

The OSF DCE **chpass** utility is not supported by Gradient DCE for Tru64 UNIX. The three commands noted previously provide most functions of **chpass**, which is a platform-dependent tool that was originally intended only as a reference implementation.

1.7.3 DIGITAL X.500 Restrictions

Version 3.1 of the DIGITAL X.500 Directory Service programming interface to the XDS and XOM APIs does not support the Basic Directory Contents Package (BDCP).

The BDCP permits certain X.500 attribute values to be expressed as OM objects. Applications using Compaq's implementation of XDS must instead specify the values of the X.500 attribute supported in the BDCP as ASN.1 BER.

An important instance of this restriction occurs when representing attribute values that are Distinguished Names, for example, when creating an alias entry with the **ds_add_entry( )** routine. You can still create aliases, but not as described in the *OSF DCE Application Development Guide*.

The *OSF DCE Application Development Guide* states that, to add an alias entry, these two attributes are required in the list of X.500 attributes specified in the Entry argument to **ds_add_entry( )**:

```
DS_A_OBJECT_CLASS = DS_O_ALIAS
DS_A_ALIASED_OBJECT_NAME = alias-target-name
```

When the BDCP is supported, the alias-target-name value is a DS-DN OM object. The syntax of this value in the Attribute-Value OM Attribute is OM_S_OBJECT.

The current DIGITAL X.500 Directory Service product requires that the alias-target-name value be supplied as ASN.1 BER, with a syntax of OM_S_ENCODING_STRING in the Attribute-Value OM Attribute.

To encode the alias-target-name from a DS-DN OM object to ASN.1 BER, complete the following steps:

1 If the DS-DN OM object is public, make it a OM private object by calling **om_create( )** and **om_put( )** routines.

2 Call the **om_encode( )** routine to convert the DS-DN OM private object to an Encoding OM object. Specify 'OM_BER' as the 'Rules' parameter to **om_encode( )** routine. The ASN.1 BER value is returned in the Encoding OM private object.

3 Call the **om_get( )** routine to extract the value. This value can be passed as the alias-target-name.

You must do the same thing when supplying member names for groups. In this case, the Entry argument to the **ds_add_entry( )** routine requires these attributes.

```
DS_A_OBJECT_CLASS = DS_O_GROUP_OF_NAMES
DS_A_MEMBER = member-name
```

The member-name value must be supplied as an ASN.1 BER value.

This encoding requirement applies whenever a Distinguished Name is input as either an attribute value to the **ds_compare( )** routine, or as a value to be added to or removed from the **ds_modify_entry( )** routine.

Whenever a Distinguished Name is returned as an attribute value from **ds_read( )**, the application must do the following:

1 Create an Encoding OM private object containing the value to be decoded by calling **om_create( )** and **om_put( )** routines.

2 Call the **om_decode( )** routine to convert the Encoding OM private object to a DS-DN OM private object.

Entegrity's implementation of the XDS API **om_encode( )** and **om_decode( )** routines supports only DS-DN OM objects. Other types of OM objects are not supported.

The following example shows how to create an alias without the BDCP. The example illustrates the steps for performing a synchronous Add Entry operation.

NOTE: The Invoke_id argument is not needed and is set to **NULL**. The workspace and bound session arguments are assumed to have been set up previously.

The entry adds the following string as an alias entry in the CDS namespace:

```
/C=US/O="Compaq Computer Corporation"/OU=Research/Projects/CDS
```

This entry, in turn points to the entry

```
/C=US/O="Compaq Computer Corporation"/OU=Research/Projects/DECdns
```

This example shows the additional steps required to supply the attribute value for the DS_A_ALIASED_OBJECT_NAME attribute if the Basic Directory Contents Package (BDCP) optional XDS package is not supported.

```
OM_private_object bound_session;   /* Assumed to be externally set up */
OM_workspace      workspace;   /* Assumed to be externally set up */


{
    OM_private_object name, alias, alias_enc_obj;
    OM_public_object  alias_enc_string;
    OM_value_position desc_count;


    OM_descriptor     cpub_dn[7];
    OM_descriptor     cpub_rdn0[3];
    OM_descriptor     cpub_rdn1[3];
    OM_descriptor     cpub_rdn2[3];
    OM_descriptor     cpub_rdn3[3];
    OM_descriptor     cpub_rdn4[3];
    OM_descriptor     cpub_ava0[4];
    OM_descriptor     cpub_ava1[4];
    OM_descriptor     cpub_ava2[4];
    OM_descriptor     cpub_ava3[4];
    OM_descriptor     cpub_ava4[4];


    OM_descriptor     cpub_attr_list[4];
    OM_descriptor     cpub_attr1[4];
    OM_descriptor     cpub_attr2[4];
    OM_descriptor     cpub_context[3];


    OM_return_code    ds_status = DS_SUCCESS;
    OM_return_code    om_status = OM_SUCCESS;


    /* Define the name AVA descriptors */


    OMX_CLASS_DESC(     cpub_ava0[0], DS_C_AVA);
    OMX_ATTR_TYPE_DESC( cpub_ava0[1], DS_ATTRIBUTE_TYPE,
                                      DSX_TYPELESS_RDN);
    OMX_ZSTRING_DESC(   cpub_ava0[2], OM_S_PRINTABLE_STRING,
                                      DS_ATTRIBUTE_VALUES,
                                      "CDS");
    OMX_OM_NULL_DESC(   cpub_ava0[3]);



    OMX_CLASS_DESC(     cpub_ava1[0], DS_C_AVA);
    OMX_ATTR_TYPE_DESC( cpub_ava1[1], DS_ATTRIBUTE_TYPE,
                                      DSX_TYPELESS_RDN);
    OMX_ZSTRING_DESC(   cpub_ava1[2], OM_S_PRINTABLE_STRING,
                                      DS_ATTRIBUTE_VALUES,
                                      "Projects");
    OMX_OM_NULL_DESC(   cpub_ava1[3]);
```

```
OMX_CLASS_DESC(     cpub_ava2[0], DS_C_AVA);
OMX_ATTR_TYPE_DESC( cpub_ava2[1], DS_ATTRIBUTE_TYPE,
                                  DS_A_ORG_UNIT_NAME);
OMX_ZSTRING_DESC(   cpub_ava2[2], OM_S_PRINTABLE_STRING,
                                  DS_ATTRIBUTE_VALUES,
                                  "Research");
OMX_OM_NULL_DESC(   cpub_ava2[3]);

OMX_CLASS_DESC(     cpub_ava3[0], DS_C_AVA);
OMX_ATTR_TYPE_DESC( cpub_ava3[1], DS_ATTRIBUTE_TYPE,
                                  DS_A_ORG_NAME);
OMX_ZSTRING_DESC(   cpub_ava3[2], OM_S_PRINTABLE_STRING,
                                  DS_ATTRIBUTE_VALUES,
                                  "Compaq Computer Corporation");
OMX_OM_NULL_DESC(   cpub_ava3[3]);

OMX_CLASS_DESC(     cpub_ava4[0], DS_C_AVA);
OMX_ATTR_TYPE_DESC( cpub_ava4[1], DS_ATTRIBUTE_TYPE,
                                  DS_A_COUNTRY_NAME);
OMX_ZSTRING_DESC(   cpub_ava4[2], OM_S_PRINTABLE_STRING,
                                  DS_ATTRIBUTE_VALUES,
                                  "US");
OMX_OM_NULL_DESC(   cpub_ava4[3]);

/* Define the name RDN descriptors */

OMX_CLASS_DESC(     cpub_rdn0[0],       DS_C_DS_RDN);
OMX_OBJECT_DESC(    cpub_rdn0[1],       DS_AVAS, cpub_ava0);
OMX_OM_NULL_DESC(   cpub_rdn0[2]);

OMX_CLASS_DESC(     cpub_rdn1[0],       DS_C_DS_RDN);
OMX_OBJECT_DESC(    cpub_rdn1[1],       DS_AVAS, cpub_ava1);
OMX_OM_NULL_DESC(   cpub_rdn1[2]);

OMX_CLASS_DESC(     cpub_rdn2[0],       DS_C_DS_RDN);
OMX_OBJECT_DESC(    cpub_rdn2[1],       DS_AVAS, cpub_ava2);
OMX_OM_NULL_DESC(   cpub_rdn2[2]);

OMX_CLASS_DESC(     cpub_rdn3[0],       DS_C_DS_RDN);
OMX_OBJECT_DESC(    cpub_rdn3[1],       DS_AVAS, cpub_ava3);
OMX_OM_NULL_DESC(   cpub_rdn3[2]);

OMX_CLASS_DESC(     cpub_rdn4[0],       DS_C_DS_RDN);
OMX_OBJECT_DESC(    cpub_rdn4[1],       DS_AVAS, cpub_ava4);
OMX_OM_NULL_DESC(   cpub_rdn4[2]);

/* And now the Distinguish Name descriptor list */

OMX_CLASS_DESC(     cpub_dn[0],         DS_C_DS_DN);
OMX_OBJECT_DESC(    cpub_dn[1],         DS_RDNS, cpub_rdn4);
OMX_OBJECT_DESC(    cpub_dn[2],         DS_RDNS, cpub_rdn3);
OMX_OBJECT_DESC(    cpub_dn[3],         DS_RDNS, cpub_rdn2);
```

```
OMX_OBJECT_DESC(    cpub_dn[4],         DS_RDNS, cpub_rdn1);
OMX_OBJECT_DESC(    cpub_dn[5],         DS_RDNS, cpub_rdn0);
OMX_OM_NULL_DESC(   cpub_dn[6]);

/* define the first entry attribute descriptor */

OMX_CLASS_DESC(     cpub_attr1[0], DS_C_ATTRIBUTE);
OMX_ATTR_TYPE_DESC( cpub_attr1[1], DS_ATTRIBUTE_TYPE,
                                   DS_A_OBJECT_CLASS);
OMX_ATTR_TYPE_DESC( cpub_attr1[2], DS_ATTRIBUTE_VALUES,
                                   DS_O_ALIAS);
OMX_OM_NULL_DESC(   cpub_attr1[3]);

/* define the second entry attribute descriptor */

OMX_CLASS_DESC(     cpub_attr2[0], DS_C_ATTRIBUTE);
OMX_ATTR_TYPE_DESC( cpub_attr2[1], DS_ATTRIBUTE_TYPE,
                                   DS_A_ALIASED_OBJECT_NAME);
OMX_STRING_DESC(    cpub_attr2[2], OM_S_ENCODING_STRING,
                                   DS_ATTRIBUTE_VALUES,
                                   NULL, 0); /*Do dynamic fix later*/
OMX_OM_NULL_DESC(   cpub_attr2[3]);

/* and now the attribute descriptor list */

OMX_CLASS_DESC(     cpub_attr_list[0],   DS_C_ATTRIBUTE_LIST);
OMX_OBJECT_DESC(    cpub_attr_list[1],   DS_ATTRIBUTES, cpub_attr1);
OMX_OBJECT_DESC(    cpub_attr_list[2],   DS_ATTRIBUTES, cpub_attr2);
OMX_OM_NULL_DESC(   cpub_attr_list[3]);

/* create the OM private object: name */

om_status = om_create(DS_C_DS_DN, OM_FALSE, workspace, &name);

/* Copy the attribute list from the cpub_dn public object into  */
/* the name private object */

om_status = om_put(name, OM_REPLACE_ALL, cpub_dn, 0,0,0);

/* create the OM private object: alias */

om_status = om_create(DS_C_DS_DN, OM_FALSE, workspace, &alias);

/* For brevity in this example we reuse the cpub_dn public object */
/* for the alias target name by fixing up one of its descriptors. */

OMX_ZSTRING_DESC(   cpub_ava0[2], OM_S_PRINTABLE_STRING,
                                  DS_ATTRIBUTE_VALUES,
                                  "DECdns");
```

```
    /* Copy the attribute list from the cpub_dn public object into    */
    /* the alias private object. */

    om_status = om_put(alias, OM_REPLACE_ALL, cpub_dn, 0,0,0);

    /* Additionally encode the alias private object */

    om_status = om_encode(alias, OM_BER, &alias_enc_obj);

    /* Extract the actual encoding string from the encoded object     */

    om_status = om_get(alias_enc_obj, OM_NO_EXCLUSIONS, 0, OM_FALSE,
                       0, 0, &alias_enc_string, &desc_count);

    /* create the OM private object: entry */

    om_status = om_create(DS_C_ATTRIBUTE_LIST, OM_FALSE, workspace,
                          &entry);

    /* Fixup the cpub_attr_list to hold the encoding string */

    OMX_STRING_DESC(cpub_attr2[2], OM_S_ENCODING_STRING,
                                   DS_ATTRIBUTE_VALUES,
                               alias_enc_string->value.string.elements,
                               alias_enc_string->value.string</ul>ngth);

    /* Copy the attribute list from the cpub_attr_list public */
    /* object into the entry private object */

    om_status = om_put(entry, OM_REPLACE_ALL, cpub_attr_list,
                       0, 0, 0);

    /* Call the Add Entry function using entry as a parameter */

    ds_status = ds_add_entry(bound_session, DS_DEFAULT_CONTEXT, name,
                                                            entry, NULL);
    if (ds_status == DS_SUCCESS)
    {
        printf("ADD ENTRY request was successful\n");
    }
    else
    {
        printf("ADD ENTRY request failed\n");
    }
}
```

CHAPTER 2

Interoperability and Compatibility

2

2.1 Overview of Compatibility with Other DCE Systems

Gradient DCE for Tru64 UNIXis based on OSF DCE Release 1.2.2. This product provides source-level runtime compatibility with DCE systems from other vendors for applications that conform to the OSF DCE Application Environment Specification (AES).

2.2 Overview of Interoperability with Other DCE Systems

Gradient DCE for Tru64 UNIX provides interoperability with DCE systems from other vendors as long as the implementations of DCE on those systems conform to the OSF DCE Application Environment Specification (AES).

2.3 DCE DFS Interoperability and Compatibility

Gradient DFS for Tru64 UNIX is a 64-bit implementation of OSF DCE Release 1.2.2. DFS, capable of supporting fileset sizes larger than 2 GB. It is compatible with the Cray 64-bit implementation of DFS.

Because most 32-bit systems cannot handle fileset sizes larger than 2 GB, operations involving these large filesets can produce unpredictable results. Take measures to prevent fileset interactions between 32-bit and 64-bit file servers in your environments. One approach is to avoid mixing 32-bit and 64-bit file servers in your environments.

2.4 CDS and DECnet/OSI DECdns Compatibility

The Compaq Distributed Name Service (DECdns) and the DCE Cell Directory Service (CDS) can coexist in a DECnet/OSI network, but they cannot interoperate. You can run both CDS and DECdns servers on the same machine, and you can build applications that make calls to both APIs. The CDS and DECdns libraries are maintained separately, as are the namespaces.

2.5 Interoperability with DECnet Phase IV and DECnet/OSI

To use Gradient DCE for Tru64 UNIX over DECnet/OSI, you must use DECnet in Phase IV compatibility mode. DECnet Phase IV compatibility mode consists of assigning a Phase IV node address to your system. A Phase

IV-compatible address is a DECnet/OSI address that falls within Phase IV limits: the area number is less than 63, and the node ID number is less than 1023. For a complete explanation of Phase IV compatibility mode, refer to the *DECnet-ULTRIX Installation and Transition Guide*.

Before you start or stop DECnet/OSI, you should first stop the DCE services. Then, after you start or stop DECnet, restart the DCE services. Use the dcesetup command **clean**, as described in the system configuration chapter, to stop the DCE services.

Enter the following command sequence to stop and start DECnet and DCE Services.

- dcesetup clean
- /etc/decnetshutdown or /etc/decnetstartup
- dcesetup start

You also have to stop and restart any DCE server applications that are running.

2.6 Interaction Between DCE DTS and DECnet/OSI DECdts

When you install Gradient DCE for Tru65 Unix, DTS is automatically installed. Normally, DTS synchronizes system clocks with other systems that use the RPC transport. The RPC transport runs on Tru64 UNIX and on other DCE systems. In addition, you can choose to synchronize system clocks with hosts running DECnet/OSI DECdts. In this section, we refer to the DTS that runs on Tru64 UNIX as DCE DTS.

The benefit of allowing DCE DTS to synchronize with DECnet/OSI DECdts is that the DECnet/OSI DECdts servers can be connected to time sources to which the DCE DTS servers do not have access. In this way, resources can be shared across a network.

One drawback to this scheme stems from DECnet/OSI DECdts servers using DECnet protocols to communicate with other DTS entities, such as servers and clerks. These protocols provide a less secure environment than the RPC protocol because the servers are unauthenticated. For example, any node can become a DECnet/OSI DECdts server at any time and could maliciously broadcast invalid times to other DECnet/OSI DECdts servers. Servers using the DECnet protocols accept and propagate this time around the network. Servers using RPC do not accept time from a server unless the server's authenticity is verified.

If your network must use authenticated connections, do not allow DCE DTS entities to accept time from DECnet/OSI DECdts servers. If your network can tolerate a small security risk, then consider allowing this interoperation.

When you answer **y** to the following configuration question, you are accepting time from DECnet/OSI DECdts servers:

```
Should this node accept time from DECnet/OSI DECdts servers? (y/n) [n]:
```

To verify whether your node is accepting time from DECnet/OSI DECdts servers, enter the following command:

```
% dtscp show all characteristics
```

Look for the command output line that says:

```
DECnet Time Source                    = FALSE
```

If you want to allow a DCE DTS entity to accept time from DECnet/OSI DECdts servers after you have configured the cell, you must reconfigure or use the **dtscp set decnet time source** command, as follows:

```
% dtscp set decnet time source true
```

To prevent a DCE DTS entity from synchronizing with DECnet/OSI DECdts servers, you must reconfigure or use the **dtscp set decnet time source** command, as follows:

```
% dtscp set decnet time source false
```

NOTE: **False** is the default value for the **decnet time source** attribute.

The nodes on your network can have different DECnet time source settings. For example, you may want to allow some DCE DTS clerks to accept time from DECnet/OSI DECdts servers, while ensuring that other DTS clerks receive time from DCE DTS servers only. This scheme works because DTS clerks receive time, but they do not propagate time to other DTS entities.

However, if even one DCE DTS server can accept DECnet/OSI DECdts time, the DECnet time eventually propagates to other DTS entities throughout the cell. The result is the same as allowing all DCE DTS entities in a cell to accept DECnet/OSI DECdts time.

DCE DTS servers also give time to DECnet/OSI DECdts clients. If you have three or more Compaq DCE DTS servers and a DECnet/OS DECdts client on your LAN, ensure that either the DCE DTS servers have access to a time provider or that at least one DCE DTS server has the **decnet time source** attribute set to **true** wherever the DECnet environment has access to a time provider. Otherwise, the three DCE DTS servers do not have an accurate time base and can give incorrect time information to a DECnet client.

You can use the DTS **dtscp show** command to display the names and values of the following specified attribute groups:

- Any local servers on a LAN segment.
- Any DECnet local servers on a LAN segment.
- Any DCE local servers on a LAN segment.
- Any global servers in the network.
- Any DECnet global servers in the network.
- Any DCE global servers in the network.
- The courier role of your server for both DCE DTS and DECnet/OSI DECdts environments.
- Whether your server is running as a DECnet/OSI DECdts courier.
- Whether your server is running as a DCE DTS courier.

If you have DECnet/OSI installed on your system, you can also use the DECnet/OSI NCL commands to manage the DCE DTS.

2.6.1 Changing the Default for DCE DTS to RPC

DCE DTS is installed by the **dcesetup** configuration program at system startup. Gradient DCE for Tru64 UNIX uses RPC to transport timing synchronization. Former versions used DLI, a feature of DECnet. To return to accepting time synchronization on DLI, you can change the default value in the **dcesetup** configuration program or you can issue a **dstd** command with the new **-m** option to override the default:

dtsd -m Accept time synchronization messages on DLI only.

DLI (Data Link Interface) is a more specific reference than DECdts. DECdts can communicate through DECnet and DLI. DLI is narrower in meaning and not synonymous with DECdts.

CHAPTER 3

Security Integration Architecture

3

3.1 Overview of SIA

The Security Integration Architecture (SIA) is a framework that supports multiple security mechanisms on Tru64 UNIX. All configured security mechanisms that run on the Tru64 UNIX operating system run under the SIA. The SIA allows you to layer various local and distributed security mechanisms onto Tru64 UNIX with no modification to either the security-sensitive commands (such as **login**, **su**, and **passwd**), or the application programming interface (API) routines that obtain password or group entries, particularly **getpwnam** and **getgrgid**.

The Tru64 UNIX operating system provides two local security mechanisms: Berkeley Standard Distribution (BSD) security and C2 class security. The default Tru64 UNIX configuration has BSD security enabled.

DCE Security, provided by OSF DCE, is a distributed network security service based on secret-key technology. It provides secure communications (authentication and data protection) and controlled access to resources (authorization) in the distributed environment. Within a DCE cell, DCE registry databases on the Security server nodes (where the Security server daemons, **secd**, run) contain information about principals, groups, organizations, accounts, and so forth. The local system administrator can create a DCE password override file, **/opt/dcelocal/etc/passwd_override**, to exclude people from using the local machine, to establish a local root password, or to tailor the local user environments.

3.2 Benefits of SIA

The local system administrator can configure DCE security to use the local SIA facility. Doing so permits users to establish a local terminal session and DCE credentials with a single login command and password.

Enabling DCE SIA also integrates DCE security with the local security mechanism. In addition, Tru64 UNIX account-related functions (such as **getpwent**) can display information from both the local and DCE registries.

DCE SIA allows you to use the DCE registry as the sole repository for all user accounts. This makes account maintenance easier because there is one central DCE registry to manage. This means that you can log in to any DCE client system if you have a DCE account, even if you do not have a local account on that system.

When a DCE system is initially configured with SIA support, you may want to use the **passwd_import** utility to migrate local accounts to the DCE registry. Account information stored on a local system in **/etc/passwd** and **/etc/group** can selectively be merged into the DCE registry. The **passwd_import** command with the check option, **-c**, displays a listing of the differences between the local registry and the DCE registry. After the command is executed, the system administrator can decide what data to merge. Having run **passwd_import** to create a brand new account in the DCE registry, you must then modify the account with **rgy_edit** to assign a password and then enable the account for DCE logins. Refer to the *OSF DCE Administration Guide — Core Components* for more information on **passwd_import**.

3.3 Using SIA

The local security mechanism for Tru64 UNIX always uses SIA. The system administrator can select DCE SIA when the machine is configured in a DCE cell. The **dcesetup** utility asks the administrator whether DCE SIA should be enabled. The default response is to enable SIA. See the *Gradient DCE for Tru64 UNIX Installation and Configuration Guide* for further instructions. Once you have configured a machine in a DCE cell, you can run **dcesetup** at any time to enable or disable DCE SIA.

When **dcesetup** enables DCE SIA, it modifies a system SIA configuration file, **/etc/sia/matrix.conf**. File **matrix.conf** is a function dispatch table used by all Tru64 UNIX security-related commands (such as **login** and **su**). Security commands call subcommands, which are in turn dispatched to one or more security mechanisms that appear in the table. Users should not modify **matrix.conf**, which is a text file, unless they need to implement a highly specialized security mechanism.

When you enable or disable SIA using **dcesetup**, you should reboot your machine. This ensures that account attribute inconsistencies are not introduced into system daemons and applications running at the time this DCE modification takes place.

When SIA is enabled and one or more servers in the cell are reconfigured, the client systems must also be reconfigured. Otherwise, DCE services cannot operate.

3.4 Using the SIA Configuration Program

The **SIACFG** configuration program helps system administrators to manage their SIA environments. Administrators can use **SIACFG** to display and resolve inconsistencies between UNIX account information stored in the DCE/Kerberos user database and corresponding information stored in the user database of the local machine.

SIACFG aids administrators in an environment where DCE SIA has been selected to control system logon based on shared/network user configuration data (that is, passwords, uids, group memberships, and gids), and where the administrator may not have the option of discarding an existing local user account database, replete with its own uid assignments, group memberships, etc.

The basic function of **SIACFG** is to compare each account and group defined in the local user database to determine if an entity with the same name and type exists in the DCE registry. If a match is found, a more detailed comparison is performed to determine if there are any inconsistencies (for example, same name but different uids, or same name but different membership lists) between the corresponding entities.

When an inconsistency is detected, **SIACFG** provides the administrator with an opportunity to resolve the conflict by designing an override entry. (See **passwd_override(5sec)** in *OSF DCE Command Reference* for a description of overrides and their format.)

When a local account or group has no analogous entity in the registry, the administrator will be asked to choose whether the account or group should have local-only or cell-wide/public significance. If **local-only** is chosen, **SIACFG** will add an override entry to the plan. When published, the plan will insure that the local entity remains distinct from any equivalently named and typed entity that might be added to the DCE registry. If the administrator chooses **cell-wide**, **SIACFG** will add directions to the plan to create a corresponding entity in the DCE registry.

For information on activating **SIACFG**, see the *Gradient DCE for Tru64 UNIX Installation and Configuration Guide*.

3.5 How DCE Security Affects the Security-Sensitive Commands and Routines

Enabling DCE SIA affects three areas of security:

- Login-related commands
- Registry information change commands
- Registry information inquiry routines

The following sections discuss these commands and routines.

A log file (**/opt/dcelocal/var/adm/security/sialog**) briefly logs the history of executing DCE SIA routines and the configured security mechanisms. You can use this file for troubleshooting or diagnosing security-related problems. You should clean up this file occasionally.

3.5.1 Login-Related Commands

SIA isolates the security-sensitive commands (**login**, **su**, **ftp**, **xdm**, **lock**, **dxsession**, **telnet**, **rtools**, and **dtools**) from the specific security mechanisms which eliminates the need to modify them for each new security mechanism. When DCE SIA is enabled in the **matrix.conf** file, SIA invokes DCE routines in response to these commands, to authenticate the login session. The following sections explain the **login** and the **su** commands in more detail. The other commands are similar to the **login** command.

3.5.1.1 login Command

When DCE SIA is enabled, using the **login** command to start a session invokes both the local security mechanism and DCE. If the login command accepts the password you enter, you have DCE credentials and are also logged in to the local security system.

NOTE: You may be logged in to the local security system even if you have no account on the local system, or if you supplied a valid DCE password that is not your local password.

If you have identical account names in both the local BSD registry and the DCE registry, account attributes in the DCE registry override those in the local registry. In other words, DCE account attributes supersede attributes in **/etc/passwd** and **/etc/group**.

If, during login, DCE security rejects your password, the local security mechanism still attempts to validate the login. If your name is in the local account database and your password is valid, you are logged in to the local system, but without DCE credentials; otherwise, the login attempt fails.

Once you have successfully logged in to DCE, a shell variable called **KRB5CCNAME** is created to point to the current session's login context and ticket cache. This variable points to a filename, such as the following:

/opt/dcelocal/var/security/creds/dcecred_68b91c00

Help prevent misuse of credentials by requiring users to remove their credentials (by running **kdestroy**) before terminating their login sessions. If your login shell is **csh**, you can insert the following lines as part of your logout profile:

```
if (‘printenv KRB5CCNAME‘ != "") then
echo "Removing login context and ticket cache..."
kdestroy
endif
```

If you run **ksh**, you can add the following line to your .profile:

```
trap 'echo Removing login context and ticket cache…; kdestroy' EXIT
```

3.5.1.2 The su Command

DCE does not support a superuser in its environment. When you issue an **su** command on a Tru64 UNIX system with DCE turned on, four results are possible. *Table 3-1* summarizes these combinations

---

NOTE: **User1** and **user2** are nonprivileged accounts.

Table 3-1: User Combinations

Combination	Current UID	New UID
1	user1	user2
2	user1	root
3	root	user2
4	root	root

The following paragraphs discuss each combination.

Combination 1: User1 to User2

You are currently **user1** and may or may not have DCE credentials. You issue an **su user2** command and are prompted for a password. The username **user2** and the password are presented to DCE Security. If they pass the security check, you get a new process with UID and DCE credentials of **user2**. If they do not pass the DCE security check (see *Section 3.5.1 on page 37*), the username and password are presented to BSD security. If they pass the BSD security check, the new process has **user2's** UID but does not have DCE credentials. If they do not pass the BSD security check, the **su** command fails.

Combination 2: User1 to Root

You are currently **user1** and may or may not have DCE credentials. You issue an **su** or **su root** command and are prompted for a password. The username **root** and the password are presented to DCE Security. Because the **passwd-override** file overrides the root account, you are not allowed to log in to DCE root. However, you get the principal credentials of the local host machine (**/.:/hosts/<hostname>/self**) after you pass the BSD security check for the **root** account.

Combination 3: Root to User2

If you only use BSD security, no password is required because **root** has the power to become anyone. This transformation is not possible in a DCE environment, because **root** (on this system) does not have any special privileges and cannot be permitted to become any principal in the cell. In this case, you are currently **root** and may or may not have DCE credentials. You issue an **su user2** command and are prompted with the following message:

```
(DCE) Enter DCE password to obtain DCE credentials,
(DCE) or press return for none.
Password:
```

The username **user2** and password are presented to DCE Security. If they pass the DCE security check, you get a new process with the UID and DCE credentials of **user2**. If they do not pass the DCE security check, or if you press **<Return>** at the prompt, the username **user2** and password are presented to BSD security. Because you are **root**, they will pass the BSD security check and the new process has **user2's** UID but does not have DCE credentials.

Combination 4: Root to Root

In this case, you are also prompted for a password, as in case 3. The username **root** and the password are presented to DCE Security. Because the **passwd-override** file overrides the root account, you are not allowed to log in to DCE root. However, because you are **root**, you pass the BSD security check, and the new process has a UID of zero and DCE credentials for the local host machine principal (**/.:/hosts/<hostname>/self**).

3.5.2 Registry Information Change Commands

The registry information change commands — **passwd** (change password), **chsh** (change shell), and **chfn** (change finger information) — change information in one of the configured security mechanism's registries. These commands invoke SIA routines, which prompt for a choice of the configured security mechanisms. In the following example, DCE SIA and BSD security are enabled.

```
You are registered with the following security mechanisms


1   DCE
2   BSD


Select ONE item by number:
```

If you choose option 1 (DCE), SIA displays the following message:

```
You have selected:
DCE
```

SIA then invokes DCE routines that allow you to make changes in the **passwd_override** file, if your requested information exists, or in the DCE registry. The system displays the following message:

```
You can change information in passwd_override or DCE registry


1  passwd_override file
2  DCE registry


Select ONE item by number:
```

The rest of the dialog is similar to the BSD dialog. When DCE SIA is enabled, there is no practical reason to maintain identical passwords in both the local and DCE security registries. A login succeeds when a valid DCE account and password are presented if the host remains configured as a DCE client and the DCE security service is available. In any event, presenting a valid local password always results in a successful local login.

The commands **passwd**, **chsh**, and **chfn** provide most functions of the OSF DCE utility **chpass**, a platform-dependent tool intended as a reference implementation from OSF. This product does not provide **chpass**.

The commands **adduser**, **removeuser**, and **vipw** do not affect the DCE registry regardless of whether DCE SIA is enabled.

Refer to the **passwd_override(5sec)** reference page in the *OSF DCE Command Reference* for details of the **passwd_override** command.

3.5.3 Registry Information Inquiry Routines

There are ten registry information inquiry routines:

- getpwent
- getpwuid
- getpwnam
- setpwent
- endpwent
- getgrent
- getgrgid
- getgrnam
- setgrent
- endgrent

These API calls also include their reentrant routines, if they exist. For these registry information inquiry routines, SIA invokes the configured security mechanism routines in the order in which they are placed in the SIA configuration file (**matrix.conf** file) until it finds the needed information. If none of the configured mechanisms fulfills the request, the **get** call fails. For example, the following line exists in the **matrix.conf** file:

```
siad_getgrnam=(DCE,/usr/shlib/libdcesiad.so), (BSD, libc.so)
```

The **getgrnam** (get a group entry by its name) call first calls the DCE **siad_getgrnam** routine to try to extract the requested information from either the **passwd_override** file, if it exists, or the DCE registry. If it succeeds, the **getgrnam** call returns the DCE information in its return group structure. If DCE fails, it continues to call the BSD **siad_getgrnam** routine to get the needed information. If it succeeds, the **getgrnam** routine returns the BSD information in its return group structure. If the **getgrnam** routine does not find the requested information in either DCE or BSD, it returns a failure; that is, a **NULL** pointer.

Operations that depend on registry information can behave more reliably when you maintain consistency between names that exist in both the local registry and the DCE registry. For instance, if a user account is registered in **/etc/passwd** and DCE, consistent password information, default shell, and similar information lets users log in using the same password even if the DCE registry is not available.

Consistency between group names in the local registry and the DCE registry is also desirable. The initial DCE cell configuration procedure declares several standard UNIX group names (such as **system**, **bin**, and **kmem**) in the DCE registry.

Because DCE SIA has precedence in the SIA matrix, a call to **getgrnam( )** for the **bin** group extracts the group attribute record from DCE rather than **/etc/group**. This has a subtle effect on such programs as **newgrp**, which call **getgrpnam( )** implicitly. In the example **% newgrp bin**, the membership list for the **bin** group is consulted to determine if the primary group of the caller

can be set to **bin**. However, **newgrp** will simply fail if membership in the **bin** group is denied in the DCE **bin** record, even though it is granted in **/etc/group**.

NOTE: The **su** command does not call **getgrnam( )** to determine membership in the **system** group. Therefore users can be granted privilege to **su root** simply by adding them to the **system** record in **/etc/group**. Also, note that the attribute values of three groups (**system**, **bin**, and **kmem**) are further qualified by their presence, by default, in the **group_override** file. See *Section 3.9.2 on page 47* for more information.

UNIX account functions such as **finger** and **getpwent** also get their attribute data from the DCE registry. Consequently, attributes such as default directory and shell must be properly maintained in the local and DCE registries. Any unintentional inconsistencies between the two registries can be troublesome. Administrators must also be careful not to inadvertently break a local account by creating an identical account name in the DCE registry for another user elsewhere on the network.

3.6 Using DCE SIA With the Tru64 UNIX Enhanced Security Option

This section explains special considerations for using the DCE SIA feature with the Tru64 UNIX Enhanced Security option.

The Tru64 UNIX Enhanced Security option, also known as C2 security, is the stricter of two security mechanisms supplied with Tru64 UNIX. The other, BSD, is supplied with the base operating system and is enabled by default. Enhanced Security is offered as an optional product, and is documented in the Tru64 UNIX Security Manual.

Enhanced Security derives much of its value from a more conservative approach toward security management. When DCE uses Enhanced Security via SIA, it operates under restrictions that affect ease of use for the DCE user and administrator. This added inconvenience is the price of a more secure system.

DCE SIA under Enhanced Security is best described by a comparison with BSD security. *Table 3-2* compares the relative SIA advantages available when the underlying security mechanism is Enhanced Security or BSD security.

Table 3-2: Benefits of Using SIA with BSD Security or Enhanced Security

SIA BSD Security	SIA Enhanced Security (C2) Option
BSD security enabled	Strict C2 security enabled
Integrated login	Integrated login (password consistency must be maintained between local and remote (DCE) registries).
Local and remote registries synchronized with **passwd_export**	Manual synchronization of local and remote (DCE) registry required. (the **passwd_export** utility is not currently available.)
Local login even without an account in the local registry	Local login requires an account in the local registry

DCE SIA offers several advantages which differ somewhat depending on whether the underlying security mechanism is BSD security or Enhanced Security.

- *Integrated login* gives a user both the local login context and the DCE network credentials simultaneously when performing a UNIX login. When the underlying security mechanism is Enhanced Security, additional administrative action is usually necessary to enable integrated login. This action will be described shortly.
- A local login to participating hosts.

While SIA offers both of these advantages with either BSD security or Enhanced Security, there are some administrative differences:

- With DCE SIA and BSD security, user account information can reside solely in the DCE Security Service registry. Account information need not be maintained in a host's local registry. This provides a significant administrative advantage as all accounts can be maintained in one convenient location rather than on separate hosts with separate login requirements. Integrated login and local login work because the local DCE SIA mechanism overrides the local registry.
- The BSD security mechanism allows the use of the **passwd_export** and **passwd_import** utilities to move account information between the local registry and the DCE registry. If account information is also maintained in the local registry, consistent passwords, UIDs, GIDs, and so on must be maintained in both the local and DCE registries.
- With DCE SIA and enhanced security, consistent user account information must be maintained in both the DCE Security Service registry and the host's local registry. *Section 3.5.3 on page 41* described the need for consistent information for routines that rely on DCE registry information. Similarly, consistent passwords must be maintained between the DCE registry and the local registry or integrated login will fail.
- The current version of the Enhanced Security mechanism does not support the use of the **passwd_export** or **passwd_import** utilities and so account information must be manually copied between DCE and participating hosts.

As with DCE SIA and BSD security, SIA does not provide an all-encompassing change password mechanism for SIA. But with Enhanced Security, the local login mechanism must succeed, so if passwords are inconsistent, integrated login is not achieved. When users run the **passwd** utility to change their passwords, the system prompts the user to select one or the other registry mechanism. The user must run **passwd** twice, to change the password in both places. If passwords are inconsistent, the user may use the local password to log in.

3.7 Performance Considerations for DCE SIA

DCE SIA is a convenience feature that greatly simplifies account administration and the acquisition of DCE credentials. However, its impact on the performance of some UNIX security functions is worth noting in certain cell and host configurations. In general, performance is constrained by a characteristic of DCE SIA-enabled systems: whenever a BSD security function is called, transparent DCE registry lookups can occur. Non-DCE applications that run on the host can perform DCE registry lookups unawares, with the result that certain functions (**getpwent( )** and **getgrent( )** in particular) may take noticeably longer.

3.7.1 Performance of getpwent( ) and getgrent( ) Functions

DCE SIA affects the speed of the **getpwent( )** and **getgrent( )** functions. When DCE SIA is not enabled, these operations result in a simple lookup in the **/etc/passwd** and **/etc/group** files. When DCE SIA is enabled, each invocation of these routines results in at least three process context switches, a bind operation to a registry (that may be remote), and a registry lookup. If performance is critical, an application or UNIX utility may need to be modified to get this data directly from the UNIX files, unless DCE registry data is specifically sought.

3.7.2 The Impact of DCE SIA on Login Performance

The UNIX login function, which is virtually instantaneous without DCE SIA, now takes from between 2 to 5 seconds. This delay affects logins to all accounts on a DCE SIA enabled host, even those that do not use the DCE registry. In most cases, users will not care because the login is a single operation performed once or twice a day.

However, some server applications, such as **ftp**, perform a system login as a matter of course and for every client connect. If a server application performs many hundreds of logins an hour, the performance of the application and the system itself may degrade noticeably.

DCE SIA is not recommended on hosts where applications make extremely heavy use of system login and registry query operations (such as **getpwent( )** and **getgrent( )**). When administrators want such applications to use the DCE registry, they can copy the DCE account and group data to the local system registry by using the DCE **passwd_export** utility. The applications can then get the DCE data from the local registry.

3.7.3 UID Management

When you create a DCE cell, you are responsible for managing the UIDs for that cell. Having incompatible UIDs between the DCE registry and the local password file is not a problem until either DCE DFS is available or DCE SIA is enabled. The initial cell creation does not use UIDs that are already in use on the local system. For its default accounts, DCE SIA uses UIDs 30 to 35 if they are available. Other DCE implementations may use accounts in the range

of 100 to 105. After cell creation, subsequent accounts in the DCE registry use the next available UID. The minimum UID value used is controlled by the **rgy_edit** command.

3.7.4 Executables in /sbin

The executables in the **/sbin** directory are statically linked. When DCE SIA is enabled, executables such as **/sbin/ls** may not properly translate UIDs and GIDs. To avoid this problem, use the executables in the **/bin** directory, which are dynamically linked with the **libc.so** shareable image. You can do this by putting **/bin** before **/sbin** in your PATH environment variable.

NOTE: The root account has **/sbin** its PATH by default.

3.7.5 rlogin

During **rlogin** to a host with DCE SIA enabled, if the incoming account has an entry in the **.rhosts** file of the target account, no DCE credentials are obtained and a warning message is displayed.

If the user's home directory is specified in the DFS namespace, access to that directory may be denied.

3.7.6 Changing root Password

If you change a machine's root password, you should run the **passwd** command twice, first to change it in the BSD location, **/etc/passwd**, then to change it in the DCE registry location, **/opt/dcelocal/etc/passwd_override**. If you change it in **/etc/passwd** but not **passwd_override**, you will see a DCE informational message, when you attempt to enter the **su** command, that says "Unable to validate/certify identify". If you change it only in **passwd_override**, the new password does not take effect.

3.7.7 Credentials Obtained for Intercell Login are Poorly Protected

When a user logs in as a principal of a foreign cell to a machine running DCE SIA (or any other integrated login system that performs a local system login on behalf of a DCE user), his or her UID identity on that machine is that of the Kerberos cross cell proxy principal created for the foreign cell. If multiple users log in to the same machine from the same cell, the local credential files created for each are owned by the same UID. As a result, users from the same cell can tamper with or borrow the credentials of others. Because exclusive control of one's credential files is an important part of DCE security, this behavior may be unacceptable for some customers running multicell environments who have strict security requirements.

NOTE: This problem does not affect the trustworthiness of intercell credentials, only those obtained during integrated login where other users from the same cell are not trusted. Administrators can disable intercell login from a foreign cell by setting the valid flag of the foreign cell principal to be not valid for login.

3.8 Performance Considerations for Registry Replication

Registry replication is a method of achieving robustness and redundancy in DCE cells. If one registry becomes unavailable, users or applications can rely on other replicas for security information. Unfortunately, replication does not always improve response, as DCE presently has no intelligent algorithm for selecting the fastest replica to bind to.

During login, the DCE runtime may choose to bind to a replica in another LAN (in the case of a multi-LAN cell), or time out waiting to connect to a replica that is not operational. With DCE SIA enabled, this behavior may present unacceptable delays in such routine UNIX shell commands as **% ls -l**.

You might consider the following guidelines and suggestions as ways to minimize replication-related latencies:

- You must actively monitor and maintain the availability of all replicas in the cell. A sound strategy is to have two hosts per LAN that serve as dedicated security replica servers, and to have a process that constantly monitors the replicas to sense their availability. Because some registry operations will bind first to a read-only replica, a multi-LAN cell should have at least one additional read-only replica on the LAN that has the master replica.
- If the cell spans LANs, be aware of latencies that may be introduced if a user in one LAN attempts to bind to and operate on a replica in another LAN. You can avoid the CDS overhead of binding to a replica, and offer a specific list of candidate replicas by inserting bindings in the **pe_site** file. This technique requires administrative attentiveness and is not generally recommended, but may be useful in some situations. Normally the **pe_site** file is used only during configuration time, to facilitate access to a registry server without reliance on CDS. It contains a hard coded list of registry bindings, one being selected randomly during a site bind. Use of this file is activated by setting the **BIND_PE_SITE** environment variable, and the effect is only for the process in which the variable is declared. A **pe_site** file is created when a host is configured, and placed in **/opt/dcelocal/etc/security**.
- Should you choose to define the **BIND_PE_SITE** environment variable, as a regular DCE management strategy, you should monitor the status of servers listed in the **pe_site** file. If a client goes to the **pe_site** file and is unable to bind successfully, the client will simply fail to bind. There is no fall-through to other registry servers that may be available.

3.9 Group Override and the group_override File

This section describes how to use the **group_override** file and its effect on local data.

3.9.1 Use of /opt/dcelocal/etc/group_override

Whenever a DCE host is initially configured, a **group_override** file is created automatically in the **/opt/dcelocal/etc** directory. This file contains override GIDs for the **bin** and **kmem** groups. Its purpose is to correct GID inconsistencies between **/etc/group** on Tru64 UNIX systems and the registry for these standard UNIX-style group names. Without this adjustment, a host running DCE SIA reports incorrect group names for these standard group UIDs with the **ls\ -l** command and produces other undesirable effects.

3.9.2 Effect of Local Override on Group Data

The new group override feature may affect local system routines, such as **groups** and **ls**, that use group name attributes. The local **group_override** file may override membership and GID attributes for group names stored in the registry. System routines such as **chgrp**, which confer access rights based on group membership, assign rights according to the following rule:

A user is given membership in a group in one of three ways:

- When granted in **/etc/group** and the group name does not appear in the registry
- When granted on the group's member list in the registry and no explicit **group_override** file entry exits to prevent it
- When a group override entry for a given group lists the user as a member

3.10 Additional Information

The following books provide more information about SIA and managing security registries:

- *DIGITAL UNIX Guide to System Administration*. This book provides a detailed explanation of SIA.
- *OSF DCE Administration Guide — Core Components*. The DCE Security Service part provides information on performing routine maintenance and importing UNIX accounts to DCE.
- *OSF DCE Command Reference*. This book contains reference pages for **passwd_import(8sec)**, **passwd_export(8sec)**, and **passwd_override(5sec)**.

CHAPTER 4

Introduction to the DCE Directory Service

4

4.1 Overview of DCE Directory Service

Distributed processing involves the interaction of multiple systems to do work that is done on one system in a traditional computing environment. One challenge resulting from this network-wide working environment is the need for a universally consistent way to identify and locate people and resources anywhere in the network.

The DCE Directory Service makes it possible to contact people and to use resources such as disks, print queues, and servers anywhere in the network without knowing their physical location. The directory service is much like a telephone directory assistance service that provides a phone number when given a person's name. Given the unique name of a person, server, or resource, it can return the network address and other information associated with that name.

The DCE Directory Service stores addresses and other relevant information as *attributes* of the name. For example, attributes can contain the name of an organizational unit, such as European Sales; a location, such as the first floor of Building A; or a telephone number. Users can search for a name by supplying one or more of its attributes. For example, given the search criteria of **John Smith** and **Chicago**, the directory service could produce a list of telephone numbers for users in Chicago named John Smith.

NOTE: Search capabilities are currently limited to the global part of the DCE Directory Service environment.

4.2 How the DCE Components Use the DCE Directory Service

The DCE Directory Service is a fundamental service that applications can rely on and use to their advantage. This section describes how other DCE components use the DCE Directory Service.

The DCE remote procedure call (RPC) interface facilitates the development and use of distributed applications that follow a client/server model. In the RPC model, clients are programs that make remote procedure calls, and servers are programs that carry out the procedures. The DCE RPC software stores information in the directory service about the addresses of RPC servers and the interfaces they support.

When an RPC client wants to make a call to a particular server, it can query the directory service for the information necessary to contact that server. If the client wants to access a specific resource that is named in the directory service, it can query for that specific name. If a client application knows the type of service that it wants, such as C compilers, printers, or employee information, but does not know the address of a specific server, it can also use the directory service to find that information.

The DCE Security Service, which verifies the identity of users when they log in, uses the directory service to store the addresses of its authentication servers.

The Distributed File Service (DFS) provides a location service for filesets (logical groups of files) so that users can access remote files as if they are on the local system. DFS uses the DCE Directory Service to find out how to contact its fileset location servers.

The Distributed Time Service (DTS) is responsible for synchronizing system clocks in the network. Synchronized clocks are important to any distributed application that needs to keep track of the order in which events occur across multiple systems. DTS uses the DCE Directory Service to find out how to locate its time servers.

4.3 How to Use DCE Directory Services

Other than DCE administrators, the people who use directory services normally do so indirectly, through an application interface. An application can interact with the directory service on behalf of users who create a name for a resource and subsequently refer to it by that name.

The following examples, both real and hypothetical, explain some of the ways that users can use the directory service:

- A user invokes a spell-checking application on a new document. The application contains DCE RPC client code on the user's local system. The RPC client contacts the directory service for information on an available spell-checking server. The directory service returns the address of the server, the protocol type it uses to communicate, and a universal unique identifier (UUID) that represents an interface. Using this information, the RPC client makes a remote call to the server and the server checks the spelling in the user's document. The user is unaware that use of the spell checker involved a call to the directory service and interaction with a remote server.
- A user logging into a system enters a name and password. The directory service helps the login program locate an authentication server, which verifies the user's identity in an authentication database.
- A user enters a file specification. The directory service provides the address of a DFS fileset location database, which contains the network address of a server that allows the user to access the file.
- A user enters the name of a computer conference or electronic bulletin board and the directory service provides an address, allowing the application to connect to the conference service.

- By entering a name or some information about a printer's capabilities, a user can learn the printer's network address. For example, the user may want to find the address of the closest and fastest available color printer.
- A user needs information from an employee in the marketing department. The user remembers that the employee's last name is Wong, but cannot remember the first name. By entering the last name and department name in an employee locator application, the user can check the directory service for information on all Wongs in the marketing department and find out how to contact the employee.
- A user enters a report in a problem-tracking database. Although the database was recently moved to a new node, the user is not aware of the change because the database is always referred to by its name only. The directory service stores the current network address and provides it to the problem-tracking application and any other application that requests it.

The remainder of this chapter explains how the DCE Directory Service environment works with regard to cells. It introduces the main directory service components: the Cell Directory Service (CDS), the Global Directory Service (GDS), and the Global Directory Agent (GDA), which is a gateway between the local and global naming environments. The chapter also discusses DCE support for the Domain Name System (DNS) and LDAP Server, which are global name services that are not parts of the DCE technology offering.

4.4 Directory Services and the Cell Environment

This section introduces the following main components of the DCE naming environment and explains their relationship to the cell:

- CDS
- GDS Client/Server
- DNS
- LDAP Client/Server
- GDA

CDS is a high-performance distributed service that provides a consistent, location-independent method for naming and using resources inside a cell (intracell). CDS can also be used for communication between cells (intercell) when cells are connected into a hierarchy.

GDS supports the global naming environment inside cells (intracell) and outside of cells (intercell). GDS is an implementation of a directory service standard known as X.500. This standard is specified by the International Organization for Standardization (ISO) 9594 and the International Telegraph and Telephone Consultative Committee (CCITT) X.500 series. Because it is based on a worldwide standard, GDS offers the opportunity for a universally interoperable global directory.

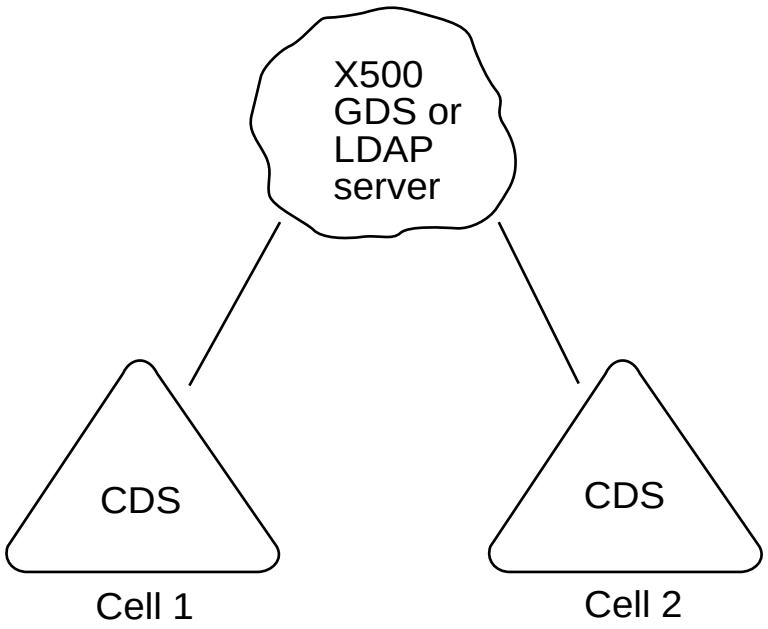
The X.500 server is a server that will accept the directory access protocol (DAP) from an X.500 client to access objects in its directory. In DCE, the server is the GDS server and the client is the GDS client. The GDA communicates with the GDS client via the XDS/XOM API. The GDS client and server are based on the 1988 X.500 standard.

The LDAP client is a client that is implemented in two libraries, **libldap.a** and **liblber.a** and they are shipped with DCE. The client is based on the University of Michigan 3.3 source code. The LDAP client accepts the LDAP API from the GDA and communicates with the LDAP server via the LDAP protocol.

The LDAP server is a server that will accept the LDAP protocol from an LDAP client to access objects in its directory. The LDAP server may be an X.500 server that also accepts the LDAP protocol or any proprietary directory service that accepts the LDAP protocol. The LDAP server is not provided by DCE and must be provided by the user. The GDA communicates with the LDAP client via the LDAP API.

Figure 4-1 represents a hypothetical configuration of two cells that each use X.500 or an LDAP server to access names in the other cell. Names that are stored directly in X.500 or the LDAP Server also are accessible from each cell. CDS is the directory service within each cell. The same organization administers both cells, which are configured based on geographic location and network topology.

Figure 4-1: Cell and Global Naming Environments



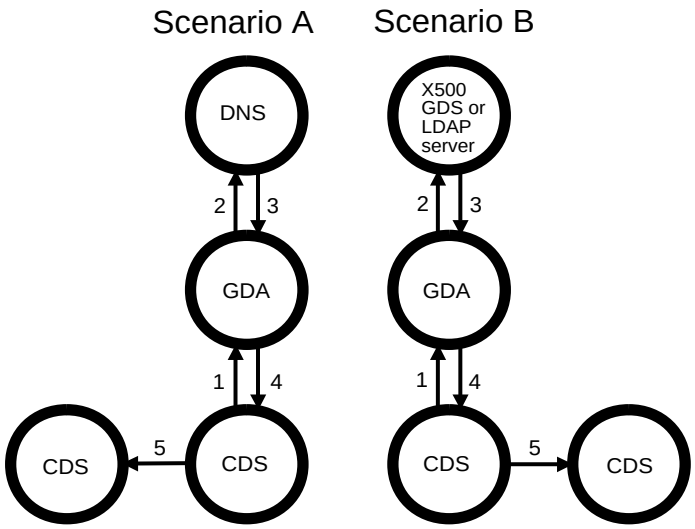
DNS is a widely used existing global name service for which DCE offers support. Many networks currently use DNS primarily as a name service for Internet host names. Although DNS is not a part of the DCE technology offering, the directory service contains support for cells to interoperate through DNS.

The GDA is the DCE component that makes cell interoperation possible. The GDA enables CDS to access a name in another cell through one of the global naming environments (X.500, LDAP, or DNS), or through the CDS of the parent cell, if the cell is part of a hierarchical cell configuration. The GDA is an independent process that can exist on a system separate from a CDS server, although by default the DCE configuration script configures the GDA on the same machine as a CDS server. CDS needs to be able to contact at least one GDA to participate in the global naming environment.

Figure 4-2 shows how the GDA helps CDS access names outside of a cell. When CDS determines that a name is not in its own cell, it passes the name to a GDA, which searches the appropriate naming environment (CDS, X.500, LDAP, or DNS) for more information about the name. The GDA returns information that enables the original CDS server to contact the CDS server in whose cell the name resides. The GDA can help CDS find names in a cell that is registered in DNS (Scenario A), a cell that is registered in an X.500 or LDAP server (Scenario B), or a cell that is registered in the originating cell's parent cell (not shown). The GDA decides which name service to use based on the syntax of the name. *Section 4.8.2 on page 58* describes name syntaxes in detail.

NOTE: The interface between the GDA and the X.500, GDS, or LDAP server is dependent on the type of server being used. The GDA uses the XDS/XOM API to interface with the GDS client. The GDS client uses the DAP protocol to interface with the X.500 Server. The GDA uses the LDAP API to interface with the LDAP client. The LDAP client uses the LDAP protocol to interface with the LDAP server.

Figure 4-2: Interaction of CDSs, GDAs, and Global Directory Services



The GDA helps CDS resolve names:
Scenario A–in another cell that is registered in DNS
Scenario B–in another cell that is registered in GDS

4.5 How Cells Determine Naming Environments

In addition to delineating security and administrative boundaries for users and resources, cells determine the boundaries for sets of names. Because different naming components operate in a cell and outside of a cell, naming conventions in the cell and global environments differ as well. The DCE naming environment supports two kinds of names: *global* names and *cell-relative*, or *local*, names. The following subsections introduce the concept of global and local names. *Section 4.8.2 on page 58* describes CDS, GDS, X.500, LDAP, and DNS names in detail.

4.5.1 Global Names

All entries in the DCE Directory Service have a global name that is universally meaningful and usable from anywhere in the DCE naming environment. The prefix **/...** indicates that a name is global. A global name can refer to an object within a cell (named in CDS) or an object outside of a cell (named in DNS), or an object outside of a cell (named in X.500).

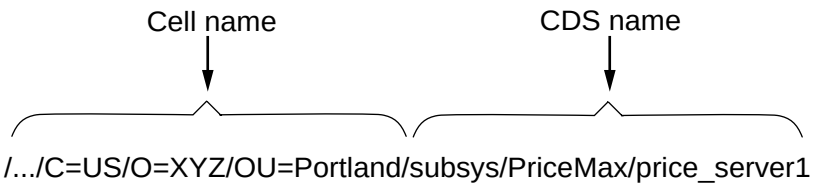
The following example shows the global name for an entry in the X.500 namespace. The name represents user Ellie Bloggs, who works in the administrative organization unit of the Widget organization, a British corporation.

```
/.../C=GB/O=Widget/OU=Admin/CN=Ellie Bloggs
```

The X.500 name syntax consists of a global prefix **/...** and a set of elements, called relative distinguished names (RDNs). Each RDN consists of one or more pairs of parts separated by an = (equal sign) character. The items that are separated by an equal sign are multiple attribute value assertions (AVAs). See the *OSF DCE GDS Administration Guide and Reference* for more information about AVAs. The first part of a pair is an abbreviation that indicates a type of information. Some common abbreviations are Country (**C**), Organization (**O**), Organization Unit (**OU**), and Common Name (**CN**). The second part of the pair is a value. (See *Section 4.9.1 on page 59* for more information on X.500 names.)

The following example shows a global name for a price database server named in CDS. The server is used by the Portland sales branch of XYZ Company, an organization in the United States.

Figure 4-3: Global Name in CDS



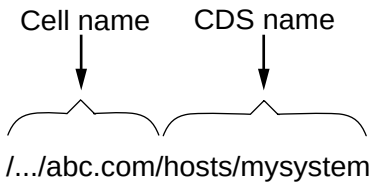
As the example illustrates, global names for entries that are created in CDS look slightly different from pure X.500 -style names. The first portion of the name, **/.../C=US/O=XYZ/OU=Portland**, is a global cell name that exists in an X.500 server or LDAP server. The remaining portion, **/subsys/PriceMax/price_server1**, is a CDS name.

The cell name exists because cells must have names to be accessible in the global naming environment. The GDA looks up the cell name in the process of helping CDS in one cell find a name in another cell. Cell names are established during initial configuration of the DCE components. Before configuring a cell that will participate in standard intercell communication (that is, the name is resolved via DNS, X.500, or LDAP server), the DCE administrator must obtain a unique cell name from either of the global naming environments, depending on whether the cell needs to be accessed through X.500 or DNS.

NOTE: The GDA transforms an X.500 cell name to the LDAP name syntax if using an LDAP server to access cell information.

The next example shows the global name of a host at ABC Corporation. The global name of the company's cell, **/.../abc.com**, exists in DNS.

Figure 4-4: Global Name Including a DNS Cell Name



4.5.2 Hierarchical Cell Names

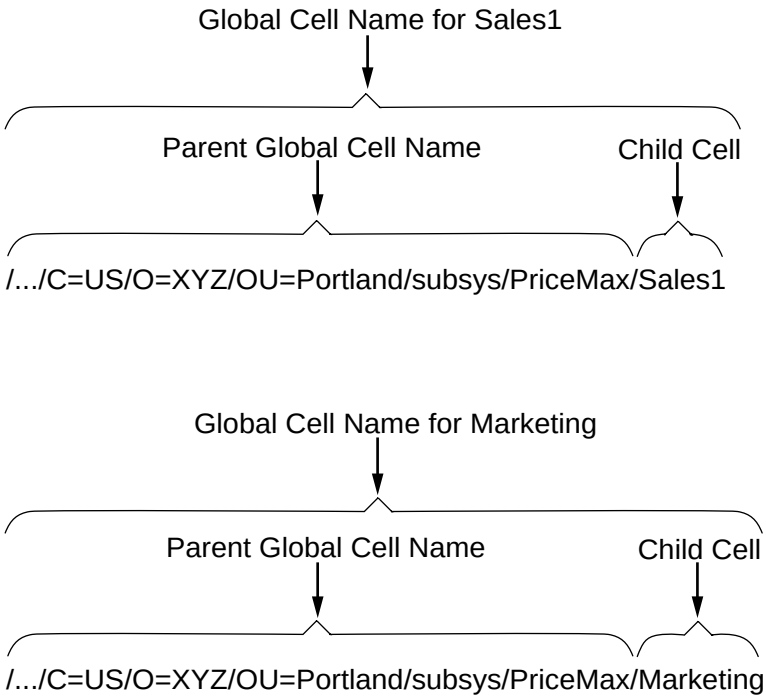
In a hierarchy of cells, the names of one or more cells, called *child cells*, are registered in a cell's CDS; this cell is called the *parent cell*. The cell at the top of the hierarchy must be registered in a global directory service (X.500, LDAP, or DNS server), but the cells underneath do not need to be since they use CDS to communicate. A child has one and only one parent at any given time, while a parent can have more than one child.

The GDA is the communications gateway between the CDS namespaces of cells in a hierarchy, as it is between CDS and the global directory services. When the GDA receives a request for information about a cell, and the cell is a child cell, the GDA returns information about the CDS in the parent cell. The CDS of the parent cell provides the pointers to the child cell.

A child cell's name begins with the parent's global cell name; that is, the name of the cell beginning at the global root **/...** prefix. (This name is also known as the parent cell's *fully qualified* name.) It ends with the specific child cell name. The parent's global name can contain CDS syntax as well as X.500 or DNS syntax, depending on where the parent cell is located in the hierarchy.

The following example shows the global cell names of two child cells:

Figure 4-5: Global Names and Child Cells



The global cell name for each child includes:

- The parent's global name, **/.../C=US/O=XYZ/OU=Portland**
- The child's unique CDS name, **/Sales1** or **/Marketing**

If a DCE administrator is establishing a hierarchy of cells during initial cell configuration, he or she must obtain a unique X.500 or DNS cell name for the cell at the top of the hierarchy from the X.500 or DNS global directory service authorities. All of the cells beneath this cell share this name. The *OSF DCE Administration Guide—Introduction* provides details on how to obtain X.500 and DNS cell names.

If a DCE administrator establishes a hierarchy of cells after the cells have been configured, the global names of the child cells change to point to the parent's cell name. The *OSF DCE Administration Guide—Core Components* provides details on how to establish a hierarchy of cells.

4.6 Alias Cell Names

You can give a cell more than one global name by creating an *alias* name for the cell. In this case, the cell has a *primary name*, which is the name that DCE services return for the cell when queried, and one or more cell aliases that the DCE services recognize in addition to the primary name. For example, if your cell is registered in the DNS global directory service, and you want to register it in X.500 as well, you obtain a X.500 name for the cell and set it up as a cell alias. The DNS name remains the primary name.

Chapter 6 of the *OSF DCE Administration Guide—Core Components* explains how to use the dcecp cellalias task object to manage your cell names. Chapter 21 of the *OSF DCE Administration Guide—Core Components* explains how to create a hierarchical cell.

4.7 Cell-Relative Naming in a Standalone Cell

In addition to their global names, all CDS entries have a cell-relative, or local, name that is meaningful and usable only from within the local cell where that entry exists. The local name is a shortened form of a global name, and thus is a more convenient way to refer to resources within a user's own cell. Local names have the following characteristics:

- They do not include a global cell name.
- They begin with the **/.:** prefix.

Local names do not include a global cell name because the **/.:** prefix indicates that the name being referred to is within the local cell. When CDS encounters a **/.:** prefix on a name, it automatically replaces the prefix with the local cell's name, forming the global name. CDS can handle both global and local names, but it is more convenient to use the local name when referring to a name in the local cell. For example, these names are equally valid when used within the cell named **/.../C=US/O=XYZ/OU=Portland**:

```
/.../C=US/O=XYZ/OU=Portland/subsys/PriceMax/price_server1

/.:/subsys/PriceMax/price_server1
```

The naming conventions required for the interaction of local and global directory services may at first seem confusing. In an environment where references to names outside of the local cell are necessary, the following simple guidelines can help make the conventions easy to remember and use:

- Know your cell name.
- Know whether a name that you are referring to is in your cell.
- When using a name that is within your cell, you can omit the cell name and include the **/.:** prefix.
- When using a name that is outside of your cell, enter its global syntax, including the **/...** prefix and the cell name.
- When someone asks for the name of a resource in your cell, give its global name, including the **/...** prefix.
- When storing a name in persistent storage (for example, in a shell script), use its global name, including the **/...** prefix. Local names (that is, names with a **/.:** prefix) are intended only for interactive use and should not be stored. (If a local name is referenced from within a foreign cell, the **/.:** prefix is resolved to the name of the foreign cell and the resulting name lookup either fails or produces the wrong name.)

4.8 Cell-Relative Naming in a Hierarchy of Cells

In a hierarchy of cells, cell-relative names and local names may not be the same. A parent cell can reference a name in a child cell by using cell-relative naming (**/.:**). Consequently, you can no longer determine whether a cell is in your local cell by merely looking at its name. In the following example, the child cell (**eng**) is named relative to its parent cell:

```
/.:/eng
```

This type of naming allows you to access names in a child cell (for example, **/.:/eng/hosts/admin**) from the parent cell, without having to specify the global name of the cell.

NOTE: When referencing names in a child cell from a parent cell, you should be mindful that your status is that of a foreign user. Therefore, the child cell may have access controls imposed on it that will deny you access to its namespace.

4.8.1 Local Filenames

When referring to pathnames of files in the local cell, you can shorten a local name even further by using the **/:** prefix. This prefix translates to the root of the cell file system. The default name of the file system root is **/.:/fs**, which is one level down from the root of the cell namespace. So, for example, the following are all valid ways to refer to the same file from within the **/.../widget.com** cell:

```
/.../widget.com/fs/smith/myfile
/.:/fs/smith/myfile
/:/smith/myfile
```

(See the *OSF DCE DFS Administration Guide and Reference* for more information on local file system abbreviations.)

4.8.2 An In-Depth Analysis of DCE Names

The rest of this chapter describes in depth the different kinds of names that make up the DCE namespace. The *OSF DCE Administration Guide—Core Components* and the *OSF DCE GDS Administration Guide and Reference* contain further details about valid characters and naming conventions in CDS, GDS, and DNS names.

4.9 CDS Names

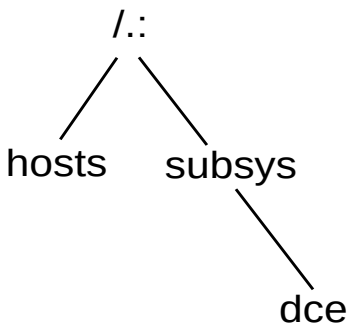
Every cell contains at least one server that is running a CDS server. A CDS server stores and maintains names and handles requests to create, modify, and look up data. The total collection of names shared by CDS servers in a cell is called a *cell namespace*. The cell namespace administrator can organize CDS

names into a hierarchical structure of directories. CDS directories, which are conceptually similar to the directories in your operating system's file system, are a logical way to group names for ease of management and use.

In a cell namespace, any directory that has a directory beneath it is considered the *parent* of the directory beneath it. Any directory that has a directory above it is considered a *child* of the directory above it. The top level of the cell namespace is called the *cell root*. You can refer to the cell root either by the global name of the cell or by the short-form **/.:** prefix.

Figure 4-6 shows a simple cell namespace hierarchy, starting at the cell root. The cell root (**/.:**) is the parent of the directories named **/.:/hosts** and **/.:/subsys**. The **/.:/subsys** directory is a child of the cell root directory and the parent of the **/.:/subsys/dce** directory.

Figure 4-6: Sample CDS Namespace Hierarchy



The complete specification of a CDS name, going left to right from the cell root to the entry being named, is called the full name. Each element within a full name is separated by a / (slash) and is called a simple name. For example, suppose the **/.:/hosts** directory, shown in *Figure 4-6*, contains an entry for a host whose simple name is **bargle**. The CDS full name of that entry is **/.:/hosts/bargle**. Multiple consecutive slashes are turned into a single slash in a full name.

Multiple directory levels enable flexibility in distributing, controlling access to, and managing many names. A directory hierarchy also reduces the probability of duplicate names. For example, the names **/.:/subsys/Hypermax/printQ/server1** and **/.:/subsys/ABC/spell/server1** are unique.

4.9.1 Names

The operation of X.500 is similar to that of CDS, but some important differences exist in the structure of names and the ways they can be looked up. Like CDS, X.500 and the LDAP Server have a server process that provides access to and management of names for X.500. This process is called a Directory System Agent (DSA). The combined knowledge of all DSAs that participate in the same global directory service implementation is called the Directory Information Base (DIB). This collective knowledge is viewed as a single global directory consisting of many entries.

Information exists in the X.500 global directory in the form of a rooted hierarchy that is called a directory information tree (DIT). The DIT is similar to a CDS namespace. However, unlike a namespace, which has no inherent rules regarding structure and content, the X.500 hierarchy is influenced by a set of rules that is called a schema. Every X.500 DSA must define a standard schema to which all of the entries in its portion of the DIB conform.

Although the X.500 standard does not mandate a specific schema, it does make general recommendations that are based largely on existing X.400 standards for electronic mail. For example, countries and organizations should be named close to the root of the DIT; people, applications, and devices should be named further down in the hierarchy. X.500 supplies a default schema that complies with these recommendations.

Every X.500 entry has a distinguished name, which uniquely and unambiguously identifies that entry. The distinguished name consists of a sequence of valid relative distinguished names (RDNs). Each RDN consists of one or more assertions of the type and value of an attribute at a particular position in the DIT. Attribute types indicate the nature of the information that is stored in the attribute value. A pair consisting of an attribute type and value is known as an attribute value assertion (AVA). RDNs can have multiple AVAs. For example, the distinguished name:

```
/C=us/O=osf/OU=branch1/CN=nollman,OU=doc-team
```

consists of four RDNs. The final RDN consists of two AVAs that are separated by a comma.

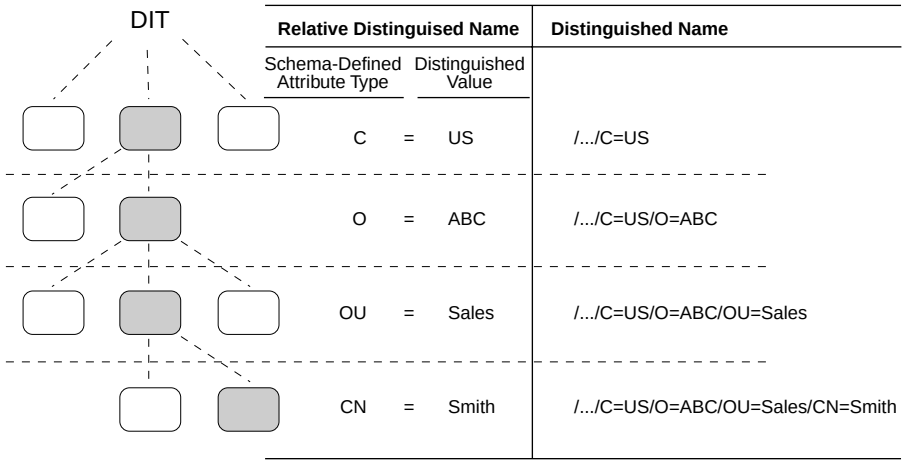
Figure 4-7 illustrates the concepts of RDNs and distinguished names and how they relate to the DIT. The figure shows the following:

- A DIT consisting of a hierarchy of schema-defined attribute types
- RDNs that result from assertions of an attribute type and value
- Distinguished names that result from a concatenation of the RDNs

An X.500 name is understood by the GDA, and it contacts either an X.500 client (GDS) via the XDS/XOM API or an LDAP client via the LDAP API to resolve the X.500 cell name.

The LDAP server contacted by the LDAP client may be proprietary or could be an X.500 server that supports the LDAP access protocol. Therefore, you may need to contact the supplier of your LDAP server for this information.

Figure 4-7: RDNs and Distinguished Names



The shaded boxes in the DIT represent the entries that are named in the column labeled relative distinguished name. The schema dictates that countries are named directly below the root, followed by organizations, organization units, and names of users. Each attribute value that makes up an RDN (and thus a distinguished name) is called a *distinguished value*.

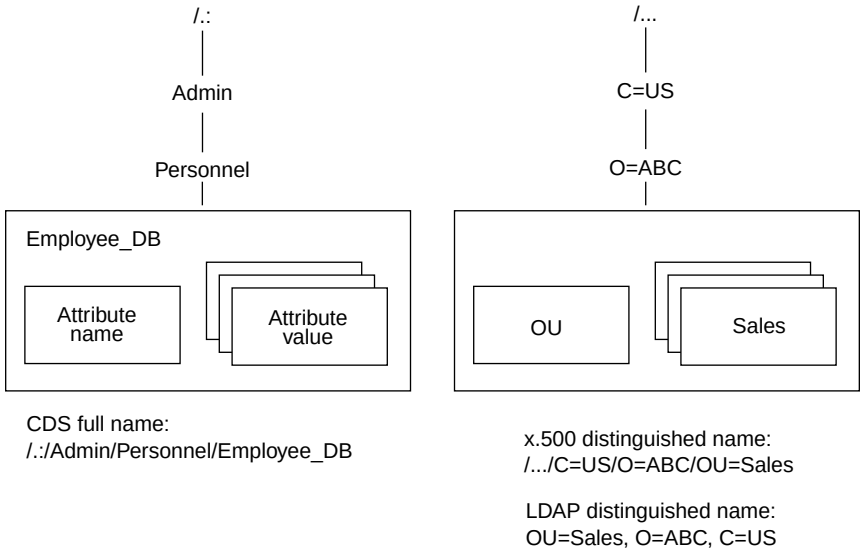
As the rightmost column in *Figure 4-7* illustrates, the distinguished name of the entry at each level of the DIT is a concatenation of RDNs from the root of the global directory to that entry's level. The lowest entry in the hierarchy, **/.../C=US/O=ABC/OU=Sales/CN=Smith**, represents the name of a user, John Smith, who works in the sales division of ABC Company, an organization in the United States. The abbreviated attribute type labels stand for Country (**C**), Organization (**O**), Organization Unit (**OU**), and Common Name (**CN**).

Figure 4-7 shows the global DCE convention for distinguished names. Each distinguished name starts with the representation of the global root (**/...**). Attribute types and values are separated by equal signs, and RDNs are separated by slashes. These conventions for specifying names are not followed by all X.500 implementations. In addition, these conventions are only used at the X.500 administration interface level. Internally, distinguished names are specified in other ways.

The structure of X.500 names points out another important difference between X.500 and CDS. A CDS name is distinct from its attributes; that is, it consists of a string of directory names ending with the simple name of the entry. In contrast, a X.500 name consists solely of a series of attribute types and their values.

Figure 4-8 illustrates this difference in the construction of CDS and X.500 names. The CDS full name **/.:/Admin/Personnel/Employee_DB** is the complete directory specification of an entry with the simple name **Employee_DB**. Attributes and their values are not a part of the CDS full name. The X.500 distinguished name **/.../C=US/O=ABC/OU=Sales** is a concatenation of attribute types and values, one from each level of a DIT schema.

Figure 4-8: Comparison of CDS and X.500 Names



NOTE: The LDAP name **/.../OU=Sales,O=ABC,C=US** is not valid in DCE. The name must be specified as an X.500 distinguished name (**/.../C=US/O=ABC/OU=Sales**).

X.500 supports the ability to search for names by supplying the values of one or more attributes. This results in what is called *descriptive naming*; in a sense, users can describe the name they are looking for. Although the search capability is valuable, it can be expensive and time consuming; so, X.500 allows users to restrict the scope of a search. Support for the search operation is limited to the X.500 environment.

4.9.2 LDAP Names

The LDAP name contains the same information as an X.500 name, but differs in its syntax. LDAP names start with the last RDN of an X.500 name and use a comma (**,**) instead of a slash (/) for RDN separators. The following example shows these differences:

```
X.500 name: /C=us/O=osf/OU=branch1/CN=nollman/OU=doc_team
LDAP name:  OU=doc_team,CN=nollman,OU=branch1,O=osf,C=us
```

DCE only supports X.500 cell names. GDA will convert an X.500 cell name to LDAP syntax when accessing an LDAP server via the LDAP client.

4.9.3 DNS Names

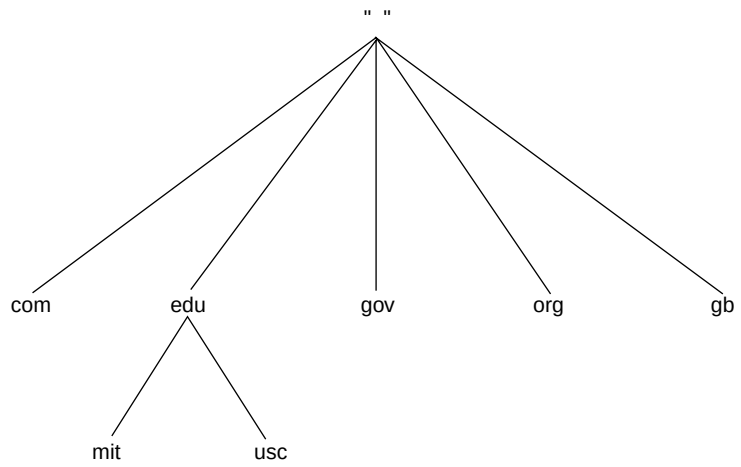
The DCE naming environment supports the version of DNS that is based on Internet Request for Comments (RFC) 1034 and RFC 1035. Many networks currently use DNS primarily as a name service for host names. The most commonly used implementation of DNS is the Berkeley Internet Naming

Domain (BIND). The BIND namespace is a hierarchical tree with its topmost levels under the control of the Network Information Center (NIC). (See the *OSF DCE Administration Guide—Introduction* for information on how to contact the NIC Domain Registrar to register a domain name.)

The names directly under the root of the BIND namespace include 2-letter codes for countries, such as **us** and **gb**, as defined in ISO Standard 3166, "Codes for the Representation of Names of Countries." Other names one level below the root include several generic administrative categories, such as **com** (commercial), **edu** (educational), **gov** (government), and **org** (other organizations). The owners of these names can grant permission to companies and organizations to create new subordinate names. *Figure 4-9* shows a sample portion of the BIND namespace. (The double quotes indicate that the root of the namespace has a null name and is not addressable.)

NOTE: Like CDS names, DNS names are not typed; that is, they do not consist of pairs of attribute types and values.

Figure 4-9: Sample Portion of the BIND Namespace



A DNS name consists of a string of hierarchical names that are separated by **.** (dots) and arranged right to left from the root of the namespace. For example, the name **ai.mit.edu** represents the branch of the namespace owned by the Massachusetts Institute of Technology artificial intelligence department.

NOTE: The order of elements in the name is the reverse of the order for CDS and GDS names.

To use a DNS cell name as part of a global DCE name, specify the DNS name intact between two slashes. For example, a cell whose DNS name is **ai.mit.edu** might contain a directory whose CDS name is **/.:/profiles**. Users should enter **/.../ai.mit.edu/profiles** to refer to the directory by its global name.

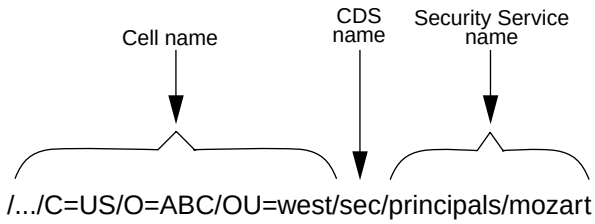
4.9.4 Names Outside of the DCE Directory Service

Not all DCE names are stored directly in the DCE Directory Service. Some services connect into the cell namespace by means of specialized CDS entries called *junctions*. A junction entry contains binding information that enables a client to connect to a server outside of the directory service.

For example, the security service keeps a database of principals (users and servers) and information about them, such as their passwords. The default name of the security service junction is **/.:/sec**.

The following example illustrates the parts of a global DCE principal name:

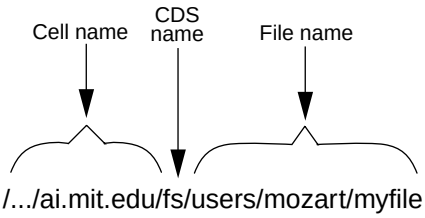
Figure 4-10: Global DCE Principal Name



The cell name, **/.../C=US/O=ABC/OU=west**, is a GDSan X.500 name. The **sec** portion is the junction entry in CDS, and **principals/mozart** is a principal name that is stored in the security service database.

Another service that uses junctions is DFS. The DFS fileset location service keeps a database that maps DFS filesets to the servers where they reside. The junction to this database has a default name of **/.:/fs**. The following example illustrates the parts of a global DCE filename:

Figure 4-11: Global DCE Filename



The global name contains a DNS cell name, **/.../ai.mit.edu**. The **fs** portion is the file system junction entry in CDS, and **/users/mozart/myfile** is the name of a file.

Thus, the DCE namespace is a connected tree of many kinds of names from many different sources. The GDA component of the directory service provides connections out of the cell and to other cells through a global namespace, such as GDS or X.500 or DNS. In a similar manner, junctions enable connections downward from the cell namespace to other services.

CHAPTER 5

Cell Directory Service Enhancements

5

5.1 Overview of CDS Directory and Clearinghouse Operations

Product Name offers some enhancements that extend the capabilities provided by OSF DCE Release 1.2.2 software. These enhancements include:

- CDS directory and clearinghouse convenience operations
- Enhanced CDS browser
- CDS Enhanced Cache Memory Control
- CDS Preferencing

CDS directory and clearinghouse convenience operations enable cell administrators to easily reorganize CDS directories and subtrees and to automate some tedious directory replication tasks.

5.1.1 Reorganizing Existing CDS Directory Replicas

After you have worked with a DCE cell for a period of time, you may observe that new CDS directories have been created in one or more clearinghouses within the cell.

When cells have multiple clearinghouses, CDS directory proliferation can cause problems or have overhead not associated with single-clearinghouse cells. For instance, at some point in the CDS directory hierarchy, master and read-only replicas of directories can become disorganized with master replicas spread among different or inappropriate clearinghouses.

For convenience in backing up your part of the namespace, you might want all of the master replicas in your part of the namespace to reside in one clearinghouse. With this strategy you need to back up a single clearinghouse because master replicas contain the most recent updates to CDS.

Gradient DCE for Tru64 UNIX provides special options (**-propagate** and **-force**) to the **directory modify** operation. These options reorganize master and read-only replicas in a CDS subtree to match the directory configuration of the subtree root directory that you name as an argument to the **directory modify** operation.

Say a directory subtree's replicas are spread among four clearinghouses (CH_A, CH_B, CH_C, and CH_D) as shown in *Table 5-1*. In the table, the master replica for **/.:/subsys/dec/srvs/vsrv** (the subtree root directory) is in CH_A. However, master replicas for its descendants reside in CH_B and CH_D. To back up the master CDS databases for this subtree, you must back up clearinghouses CH_A, CH_B, and CH_D.

You can perform a **directory modify** operation to reorganize all master and read-only directories to be in the same clearinghouse as the subtree root directory. Once all master replicas are in the same clearinghouse (CH_A in our example), you need only back up clearinghouse A.

Table 5-1: Reorganizing Existing CDS Directory Replicas

Item	Description	CH_A	CH_B	CH_C	CH_D
1	/.:/subsys/dec/srvs/vsrv (subtree root) configuration	master	r-only		r-only
2	/.:/subsys/dec/srvs/vsrv/vdat1	r-only	master		r-only
3	/.:/subsys/dec/srvs/vsrv/vdat2	r-only	r-only		master
4	/.:/subsys/dec/srvs/vsrv/vdat3	r-only	r-only	r-only	master
5	All directories in subtree	master	r-only		r-only

Items 1, 2, 3, and 4 depict the master directory organization before reorganizing them with the **directory modify** operation. Item 5 illustrates the results of the following **directory modify** operation:

```
% directory modify /.:/subsys/dec/srvs/vsrv -propagate -force
```

Note that the preceding **directory modify** operation removes the replica from clearinghouse C (CH_C) because the root directory (**/.:/subsys/dec/srvs/vsrv**) has no replicas in clearinghouse C.

5.1.2 Creating Additional CDS Directory Replicas

You can also use a **directory modify -propagate** operation to automate the manual steps formerly needed to create additional (read-only) replicas of new directories you have created. This operation creates read-only replicas of the new directory. Note that this operation also affects any siblings of the new directory and all of their descendants.

When you create a new directory or directory subtree in a CDS clearinghouse, the directory create operation creates only the master replica. By default, the replica is created in the same clearinghouse where the parent directory's master replica resides.

You can use a **directory modify -propagate** operation to create read-only replicas of the new directory or directory subtree. The new replicas will be created and organized so that their master and read-only replicas will be in the same clearinghouses as the subtree root directory (the parent directory) which you name as the argument to the **directory modify** operation.

The next table illustrates the behavior of the **directory modify -propagate** operation used for creating read-only replicas of a new directory.

Table 5-2: Creating Additional CDS Directory Replicas

Item	Description	CH_A	CH_B	CH_C	CH_D
1	Initial parent replica configuration	master	r-only		r-only
2a	Create new child directory using default	master			
2b	Use -propagate option	master	r-only		r-only
3a	Create new child directory using default		master		
3b	Use -propagate -force options	master	r-only		r-only

Item 1 shows the initial configuration of the parent directory. The master replica is in clearinghouse A (CH_A). Read-only replicas of the parent directory reside in clearinghouses B and D. Clearinghouse C does not contain any replicas of the parent directory.

Item 2a illustrates the result of the default **directory create** operation which creates the new directory in the same clearinghouse where the parent directory's master replica resides. Default means not specifying an alternative clearinghouse in which to create the new directory.

Item 2b shows the results of the **directory modify -propagate** operation which creates master and read-only directory replicas in the same clearinghouses that contain the parent's master and read-only directory replicas.

Item 3a illustrates the result of a **directory create** operation which creates the new directory in a different clearinghouse than where the parent directory's master replica resides. An optional **-clearinghouse** option to the directory create operation specifies to create the new directory in clearinghouse B.

Item 3b shows the results of the **directory modify -propagate -force** operation. Here, the **-force** option must be used. Otherwise, an error occurs because the new directory's master replica is in a different clearinghouse than the parent directory's master replica. The **-force** option performs an extra step, causing the new directory master and read-only replicas to conform to the same configuration as the parent directory replicas.

The **directory modify -propagate** operation affects all the descendant directories of the named directory. For example, assume your cell has the following subtree configuration:

```
/.:/subdir1/subdir2/newdir1
/.:/subdir1/subdir2/olddir1/oldsubdir1
/.:/subdir1/subdir2/olddir2/oldsubdir2
```

The following operation organizes the replicas of all three child directories (**newdir1** , **olddir1** , and **olddir2**) and their descendant directories to match the master and read-only replica configuration of the parent directory (**/.:/subdir1/subdir2**) which is named in the operation.

```
% directory modify /.:/subdir1/subdir2 -propagate -force
```

Of course you can have the master replicas of child directories in different clearinghouses than their parent's master replicas. However you must manually create any read-only replicas using separate **directory create** operations for each replica you want to create.

5.2 Enhanced Browser

The Browser is a Motif-based tool for viewing the CDS namespace. The Browser can display an overall directory structure as well as show the contents of directories, enabling you to monitor growth in the size and number of directories in your namespace. You can customize the Browser so that it displays only a specific class of object names. The Gradient DCE for Tru64 UNIX Enhanced Browser contains some additional functions beyond those contained in the OSF DCE Version 1.1 Browser.

5.2.1 Displaying the Namespace

When you start the Browser, an icon representing the root directory is the first item to appear in the window. Directories, soft links, and object entries all have distinct icons associated with them. Most object entries have unique icons based on their class; the class indicates the type of resource that the entry represents (for example, clearinghouse object entries). When the Browser does not recognize the class of an entry, it displays a generic icon.

The following figure shows the Enhanced Browser icons and what they represent.

Figure 5-1: Enanced Browser Icons

Icon	Entry Type
	Directory
	Object entry (generic)
	Soft Link
	Clearinghouse object entry
	Group

5.2.2 Filtering the Namespace Display

Using the Filters menu, you can selectively display object entries of a particular class. With the Enhanced Browser, you can choose from either the RPC_Class or CDS_Clearinghouse object classes. For example, if you are interested in seeing the entries for clearinghouse objects only, choose the class

CDS_Clearinghouse from the Filters menu. If you are interested in seeing object entries used in the name service interface (NSI), choose RPC_Class. You can filter only one object class at a time.

Setting a filter does not affect the current display, but when you next expand a directory, you see only object entries whose class matches the filter. Note that soft links and directories still appear because only object entries can be filtered out. To reset the filter to view all object entries, choose the asterisk (*) from the Filters menu.

For a full description of the Browser, see the CDS part in the *OSF DCE Administration Guide — Core Components*.

5.3 CDS Enhanced Cache Memory Control

Two options for the **cdscache discard** command allow administrators to control the release of memory from the cache clerk without having to stop and shut down DCE. The new options **-entry** and **-replica** specify structures in the cache for deletion. The following command shows how to delete a replica:

```
dcecp -c cdscache discard -replica /.:/foo_ch
```

where *foo_ch* should be replaced by a valid clearinghouse name.

5.4 CDS Clearinghouse Preferences

With this release, CDS is able to make more intelligent choices about which clearinghouse to contact in satisfying a user request. This has the potential of greatly improving performance, depending on your cell configuration. Each client machine ranks clearinghouses in the order in which they should be contacted by the client for CDS information. Default behavior prioritizes those located “closest” to the client on the network. However, the administrator of a client node can override the default rankings.

This enhancement is useful in situations where, for example, there are multiple high-performance LANs, each with its own CDS server, connected by a low-performance WAN. With this feature, the clearinghouse with the best ranking is the one on the machine with the server, followed by one on the same LAN with the client. Local clearinghouses are preferred over distant clearinghouses. Clients use distant clearinghouses only when local clearinghouses cannot satisfy a request. The administrator can override the defaults as needed.

Clearinghouse preferences are achieved by assigning a numeric rank to each clearinghouse. A rank is a 16-bit unsigned integer (range 0-65535). Lower numbers are preferred over higher numbers (and a rank of 65535 means “don't ever use this clearinghouse”).

To override defaults, ranks must be specified in a text file called opt/**dcelocal/etc/cds_serv_pref**. The format of the file is one clearinghouse name and one rank on each line of the file. Blank lines and comments (“#” to the end of the line) are ignored. Ranks can be 0-65535 (0x0000-0xffff) and can be specified in decimal, octal (with leading “0”), or hex (with leading “0x”). Clearinghouse names can be in any of the following formats:

```
/.../cellname/foo_ch
/foo_ch
foo_ch
/.:/foo_ch
```

If the clearinghouse's cell name is not specified, the local cell is assumed.

Example file:

```
/.:/foo_ch  50  # most preferred clearinghouse
/.:/bar_ch  100
/.../mycellname/baz_ch 100
```

If a clearinghouse is not mentioned in the preferences file, a rank is calculated for it. Thus, you need to specify rank for a clearinghouse only when you want to override its default rank.

The default ranks are calculated based on IP address:

- Clearinghouses with addresses that match the local host address get a default rank of 5000.
- Clearinghouses on the same IP subnet as the local host get a default rank of 20000.
- Clearinghouses on the same IP network as the local host get a default rank of 30000.
- All other clearinghouses get a default rank of 40000.

The clearinghouse preferences file is read upon cdsadv startup and the values are cached. If you change rank values, you must stop the CDS client, remove the cache, then restart the CDS client.

The following commands now include a rank attribute:

```
dcecp -c cdscache show -clearinghouse /.:/foo_ch
cdscp show cached clearinghouse /.:/foo_ch
```

where *foo_ch* should be replaced by a valid clearinghouse name.

CHAPTER 6

LDAP Capabilities

6

6.1 Overview of LDAP

The Lightweight Directory Access Protocol (LDAP) provides access to the X.500 directory service without the overhead of the full Directory Access Protocol (DAP). The simplicity of LDAP, along with the powerful capabilities it inherits from DAP, has made it a *defacto* standard for Internet directory services.

DCE has relied on CDS to provide both intra-cell and inter-cell directory service. Inside a cell, the directory service is accessed mostly through the name service interface (NSI), implemented as part of the runtime library. Cross-cell directory service is controlled by a global directory agent (GDA), which looks up foreign cell information on behalf of the application in either the Domain Naming Service (DNS) or X.500 database. Once that information is obtained, the application contacts the foreign CDS in the same way as the local CDS.

DCE gains LDAP support for both NSI and GDA. From an application standpoint, any application within NSI can now reach the LDAP directory service. From a GDA standpoint, GDA can now look up foreign cell information by communicating through LDAP to either an LDAP-aware X.500 directory service or a standalone LDAP directory service, in addition to DNS and DAP.

This release provides LDAP as an optional directory service that is independent of CDS. From an application standpoint, it duplicates CDS functionality. LDAP does not replace CDS as the directory service for DCE nor does it coexist with CDS Version 3.0 of DCE for Compaq Tru64 UNIX . LDAP is provided as an option for customers looking for an alternative that offers TCP/IP and internet support.

Gradient DCE for Tru64 UNIX does not automatically install LDAP. Prior to installing DCE, a DCE administrator must obtain LDAP software and install it as an LDAP server in the environment. Next, a DCE administrator must choose LDAP during the DCE installation and configuration procedure and configure LDAP directory service for a cell. Once LDAP is configured, applications can request directory services from either CDS or LDAP, or both. Whether or not LDAP is configured, DCE system processes continue to rely on CDS to provide directory service.

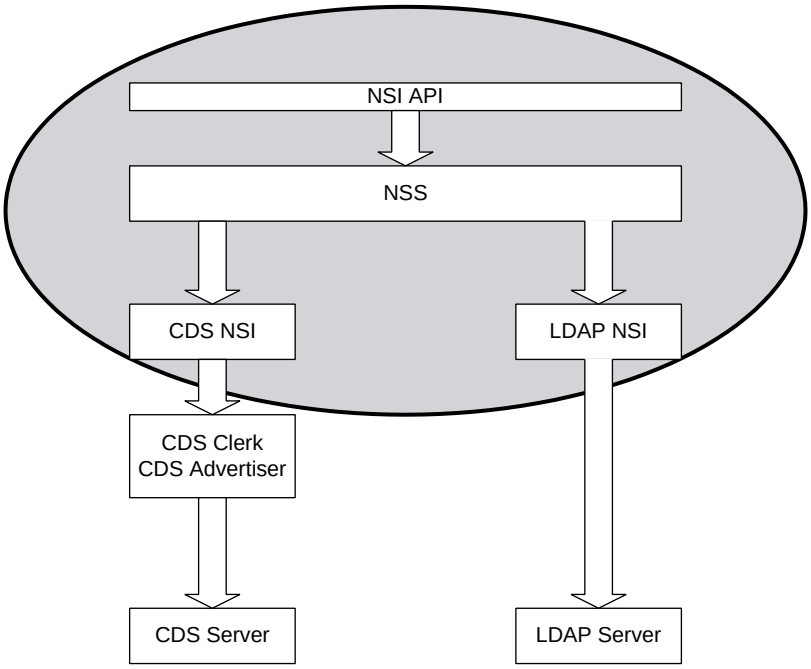
6.2 How NSI Works

NSI stores and retrieves RPC binding, group, and profile information in either the CDS directory service or LDAP, or both.

NSI implements the name service switch (NSS), which selects among configured directory services when executing an NSI call.

Exactly which name service(s) are selected by NSS depends upon the name and syntax arguments to the NSI call, the NSI runtime configuration options, and the nature of the call itself, as shown in *Figure 6-1*

Figure 6-1: NSI Architecture



6.2.1 LDAP Syntax

In addition to the CDS syntax, **rpc_c_ns_syntax_dce** (previously supported by NSI), the LDAP-enabled NSI supports a new name syntax, **rpc_c_ns_syntax_ldap**.

If this new syntax is specified in an NSI call, the corresponding name must be an LDAP Distinguished Name (DN), which NSI uses to obtain information from the LDAP directory service.

If the CDS syntax is used in an NSI call, it is not apparent from the syntax which directory service, LDAP or CDS, is to be contacted. The run-time NSI configuration options and the nature of the call join in the decision. If LDAP is selected, NSS translates the name from CDS syntax to LDAP syntax. The purpose of the syntax translator is to make LDAP accessible from applications using a CDS syntax.

6.2.2 NSI Configuration

A one-time initialization executes when an application accesses NSI for the first time. The initialization determines which name services the application wants to use and the priority of each name service. The easiest way to provide the required information is with a configuration file.

If the LDAP name service is specified, the initialization must be able to find the address of the host where the LDAP server is running, the port it is listening on, and the cell name mapping from DNS or X.500 syntax to LDAP syntax. If more than one name service is configured, the export mode that determines if updates need to be sent to all name services has to be specified.

The environment variable **RPC_CONFIG_FILE** can be used to specify a configuration file. A configuration file can specify a DNS name to query for configuration information, allowing central control of client configuration. Storing LDAP configuration information in the TXT records of a DNS name is the recommended way of configuring the LDAP NSI.

Here is the recommended way to use a configuration file for LDAP:

1 Choose a DNS name. (Consult with the DNS administrator.)

2 Store the configuration information in the TXT records of a DNS name for use before the first process that uses RPC is executed.

3 Create a file using the LDAP NSI configuration file syntax, and include all the options and values appropriate to your site, as explained in the next section.

4 Ask the DNS administrator to create a TXT record for each line in this file and to add these TXT records to the set of records belonging to the domain name you chose.

5 Use the **RPC_CONFIG_FILE** environment variable to specify the location of the file you want to be the configuration file. This file contains only a single line, specifying the target DNS name you chose previously:

```
RPC_NS_DNS_CONFIG_INFO domain_name
```

The syntax of the LDAP NSI configuration file maps easily to a set of attribute type/value pairs. The pairs are described in the next section.

6.2.3 Configuration File Format and Syntax

The configuration file contains values for various configuration options used by the NSI runtime library. Each line of the file is of the following form:

config_option_name config_option_value

If multiple values are specified for a particular configuration option separated by a tab or a space character. Each value must be specified as a separate option name/option value pair on a separate line.

The following table describes the possible values for the *config_option_name* and *config_option_value* fields in the LDAP configuration file.

Table 6-1: LDAP NSI Configuration Options and Values

config_option_name	config_option_value
RPC_NS_NAME_SERVICE	integer: l 100 CDS|LDAP An integer priority value followed by the name of a name service known to the NSI. Currently, the only known services are CDS and LDAP. Multiple name services may be configured and are tried in priority order subject to the service selection rules. Lower priority values indicate higher priority No default. At least one name service must be specified. Multiple name services may be specified, if desired. If the same name service is specified more than once, the last priority value specified is used. If more than one name service has the same priority value, it is undefined which service has (effectively) the highest priority.
RPC_NS_EXPORT_MODE	write_one|write_all Specifies whether export operations should write only to the first possible name service or all possible name services. Services are tried in priority order. Default is write_one; in the case that the option is specified multiple times, only the last-specified value is used
RPC_NS_LDAP_SERVER	*hostname_or_ip_addr*[*:port*] The name or IP address of an LDAP server to which the NSI can connect. The appropriate port number may also be specified. The default port number (389) is used if another is not specified. Multiple LDAP servers can be specified, but only the first specified server will be used.
RPC_NS_LDAP_CELLMAPPING	*dce_cell_name ldap_dn* Associates the *ldap_dn* with the *dce_cell_name so* that names in the specified cell are searched for in the specified LDAP subtree. No default. Multiple options may be specified; if multiple mappings for the same cellname are specified, the last-specified mapping is used.
RPC_NS_DNS_CONFIG_INFO	*domain_name* A domain name the NSI can query to obtain configuration information. This domain name should possess TXT records formatted exactly like lines in the configuration file: each TXT record has an initial *config_option_name* followed by white space and a *config_option_value*. No default. Multiple options may be specified; the effect is to read each specified DNS name for configuration information.

6.2.4 NSI Call Categorization

Usually, an NSI call is either read or write. A read NSI call obtains information from the directory service but makes no changes. A write call creates, deletes, or updates a directory. An example of the read NSI call is **rpc_ns_binding_lookup_next**. The call **rpc_ns_binding_export** is an example of write NSI call.

NSI calls that are neither read nor write calls are miscellaneous calls, of which an example is **rpc_ns_binding_select**.

A read NSI call completes when the information is obtained from a configured directory service. NSI does not guarantee the consistency of information between different configured name services.

A write NSI call may or may not complete when the operation succeeds in one configured directory service, depending on the export mode run-time configuration option.

6.2.5 Name Service Selection

For a read NSI call, the following pseudocode describes the NSS selection algorithm:

```
for each configured name service in the specified priority {
if the name is in the native syntax of the name service {
    or
   a translation routine exists to the native syntax {
append the name service to the trial list
}
}


if there is no name services on the trial list {
return rpc_s_name_service_unavailble
}


for each name service on the trial list {
make the call
if the call succeeds {
return success
} else if the call fails with other than rpc_s_entry_not_found {
return the error
} else if there is no more name services {
return rpc_s_entry_not_found
}
}
For a write NSI call, the following pseudocode describes the NSS selection
algorithm:
for each configured name service in the specified priority {
if the name is in the native syntax of the name service {
    or
   a translation routine exists to the native syntax {
append the name service to the trial list
}
}


if there is no name services on the trial list {
return rpc_s_name_service_unavailble
}


for each name service on the trial list {
make the call
if the call succeeds {
if export mode is to update one name service {
return success
}
```

```
} else {
return the error
}
}
```

Among the read NSI calls, contexts provide a way to maintain information across successive calls. An example of a read NSI call with context is **rpc_ns_binding_lookup_next**. The context is built in a previous **rpc_ns_binding_lookup_begin** and destroyed in **rpc_ns_binding_lookup_done**. NSS manages calls to assure that only the call to construct the context runs the selection algorithm. Successive calls bypass the algorithm and use the same selected name service.

6.2.6 Name Translation from CDS to LDAP

The NSI controls the CDS-syntax-to-LDAP-syntax translation of names. CDS-to-LDAP translation supports applications using the LDAP directory service with names based on CDS syntax. Another, larger purpose of name translation is to separate applications from dependence on particular directory services.

A configuration file controls NSI. A specific DCE cell name is associated with the DN of a subtree in the LDAP name space. The mapping of a DCE cell name in either DNS or X.500 style to the distinguished name (DN) must be provided in the NSI configuration file.

To translate a name in CDS syntax to LDAP syntax, the cell name part is translated using the mapping(s) specified in the NSI configuration file. The cell relative part is transformed with the order of the component names reversed. The component name is prefixed with **cn=**, commas are substituted for slash separators. Quoted special characters in CDS, and unquoted and unquoted special characters in LDAP, are quoted.

For example, if the mapping between /.../dce.mycompany.com and ou=dce,o=mycompany,c=us is defined, the name /.../dce.mycompany.com/foo/bar is translated as cn=bar,cn=foo,ou=dce,o=mycompany,c=us. The name /.../dce.mycompany.com/foo=bar is translated as cn=foo\=bar,ou=dce,o=mycompany,c=us. Note the handling of special characters in the second example.

6.3 Using NSI

This section describes NSI configuration issues and possible differences between CDS and LDAP.

6.3.1 Modifying Runtime Configuration Options

The NSI initialization process first checks if a value (file name) is set for the environment variable, **RPC_CONFIG_FILE**. If it finds the environment variable and the name of a file, the contents of the file is used to initialize the NSI. If no environment variable is present, the NSI initialization process looks for the default system runtime configuration file, **/opt/dcelocal/etc/rpc.conf.**

If neither the default configuration file nor the environment variable exists, NSI initialization fails with the error status **rpc_s_file_not_found**. If the file is not in the format as described earlier or is corrupted, NSI initialization fails with error status **rpc_s_invalid_file_format**.

Note that the default configuration file is very important. Modifying the default configuration file, **/opt/dcelocal/etc/rpc.conf,** affects configuration options for all DCE processes on the same host machine.

DCE installation configures CDS. DCE system processes rely on CDS to provide directory service security and reliability. Poorly-considered changes to the system defaults in the NSI configuration file can have particular consequences for the security daemon and CDS advertiser, and thereby compromise a DCE cell.

Please note that it is strongly recommended that you leave the default configuration file unchanged. Instead, use the environment variable **RPC_CONFIG_FILE** to alter runtime NSI configuration options. By providing a user-specified configuration file rather than altering the system default file, you safeguard an environment in which CDS and LDAP can continue to work properly.

6.3.2 Application Programming

For the sake of source and binary compatibility, the application programming interface (API) for the name service is unchanged. Note that an application might behave differently if LDAP is configured. A difference may result from LDAP itself or the availability of multiple name services.

A programmer must keep several things in mind:

- CDS is a directory service that has no schema support. Any kind of data can be written to any kind of entry. Although users are advised to follow certain styles, they are not required to do so. LDAP mandates the use of schema, as X.500 does. It is likely that if a program does not follow the style and it succeeds in CDS, it might fail with LDAP configured.
- Security is not supported in LDAP in Gradient DCE for Tru64 UNIX. All NSI calls are unauthenticated. The NSI call **rpc_ns_set_authn** has no effect on LDAP operations.
- NSI calls for setting expiration ages have no effect on LDAP operations as LDAP does not support caching.
- LDAP does not guarantee consistency among different directory services. Searching an entry in the LDAP directory service can return completely different results from the corresponding entry in CDS. Also, setting export mode to updating all configured directory services does not guarantee transactional behavior, which means the update procedure may succeed in one directory service and fail in another one and not try yet others.

To use multiple directory services, the understanding of NSS selection algorithms is essential.

6.3.3 NSI Known Limitations

6.3.3.1 Security

LDAP NSI offers no support for security. The lack of security makes an LDAP directory service vulnerable to spoofing or denial of service attacks.

6.3.3.2 Schema

Although CDS does not support schemas, it does support the following concepts:

- **Default Entry**—Stores bindings
- **UUID Entry**—Stores universal unique identifiers
- **Group Entry**—Stores members information
- **Profile Entry**—Stores profile elements

In LDAP, the object schema implements those same concepts and reinforces them.

As a result of the schema, certain kinds of data can only be exported to certain kinds of entries. For example, bindings cannot be exported to group entries and group members cannot be exported to profile entries. If an LDAP NSI operation is called for an incompatible kind of entry, the call fails with **rpc_s_entry_not_found** even if the entry indeed exists in the name space.

However, both CDS and LDAP support the notion of “upgrade.” Namely, one can perform operations that are permissible to a server entry on a default entry, in which case the default entry is “upgraded” to a server entry. The same applies to group entry and profile entry.

There is one exception to the rule. In CDS, it is legal to export only UUIDs to a default entry, but this is not allowed in LDAP. Because the default entry does not allow UUIDs, LDAP would have to “upgrade” it. But both server entry and group entry allow UUIDs, there is no way of knowing which type of entry to “upgrade” to. This implementation of LDAP NSI chooses to return an error in an ambiguous case like this.

6.3.3.3 Schema for Storing RPC Entries in a Directory Service

This section defines a schema that conforms closely to the DCE conceptual model for RPC entries. This schema allows an RPC NSI implementation to use LDAP to store RPC entries and to use LDAP queries to implement the RPC NSI lookup APIs.

The implementation supports three kinds of RPC Name Service Entries:

- **Server Entries—**Support the retrieval of a set of string bindings for any combination of Entry Name, Interface ID and version, Object ID, Transfer Syntax, and transfer syntax version.
- **Group Entries—**A set of RPC entries identified by an Entry name.
- **Profile Entries—**A profile establishes a priority-based search order through a set of entries. This is essentially a “list of sets” with the outer list ordered by priority and each inner set at the current priority.

DCE RPC defines the concept of a “mixed entry” in which a single entry serves multiple purposes—for example, entries that serve as both Group and Server entries. Mixed entries are not supported by this schema. This seldom-used DCE RPC feature leads to unnecessary complexity for both implementers and users of the RPC NSI.

To meet these requirements, a schema defines six object classes:

- rpcEntry
- rpcGroup
- rpcServer
- rpcServerElement
- rpcProfile
- rpcProfileElement

A schema also defines nine attribute types:

- rpcNsObJectID
- rpcNsGroup
- rpcNsPriority
- rpcNsProfileEntry
- rpcNsInterfaceID
- rpcNsAnnotation
- rpcNsCodeset
- rpcNsBindings
- rpcNsTransferSyntax

Taken together, these object classes and attributes implement the DCE-RPC concept of an entry.

The **rpcEntry** object class is the class from which all other RPC objects derive, so that they may be easily located in a search.

An **rpcGroup, rpcServer**, or **rpcProfile** object forms the “root” of an entry. The type of entry is determined by the object class. Note that the types are mutually exclusive; an entry cannot serve multiple purposes. Separating the entry types into distinct object classes, as shown in *Table 6-2*, simplifies the task of the NSI provider in determining how to handle a given entry.

Table 6-2: Entry Types and Object Groups

Entry Type	Object Class(es)
Group	**rpcGroup** holds a set of references to other **rpcEntry objects**
Profile	**rpcProfile,** a container holding a set of **rpcProfileEntry** objects, each holding a list of references to entries with a given priority
Server	**rpcServer**, a container holding a set of **rpcServerElement** objects, each holding the identification of one or more interfaces (and/or objects) offered by a given server

6.3.4 Objects and Attributes

The following treats a number of items for programmers.

6.3.4.1 Notation

The notation used in this document is the same as that used in *Lightweight Directory Access Protocol: Standard and Pilot Attribute Definitions*, with the following difference: the referenced notation does not allow the expression of both permissible parentage and class inheritance. The BNF in the cited draft for defining object classes is therefore extended as follows:

```
<ObjectClassDescription> ::= "("
<oid> -- ObjectClass ldentifier
[ "NAME" <DirectoryStrings> ]
[ "DESC" <DirectoryStrings> ]
[ "OBSOLETE" ]
[ ·"SUP" <oids> ] – ObjectClass[es] from which this class is derived
[ ·"PARENT" <oids> ] - Permissible parents of this object class
[ ( ·"ABSTRACT" | ·"STRUCTURAL" | ·"AUXILIARY" ) ]
[ ·"MUST" <oids> ] -- AttributeTypes
[ ·"MAY" <oids> ] -- AttributeTypes
")"
```

6.3.4.2 Object Naming

All objects have cn (common name) as their naming attribute; this attribute provides the RDN for the object.

6.3.4.3 Object Definitions

In addition to the object classes listed in the sections below as allowed parents, there must be at least one other object class allowed as a parent to root the tree. Furthermore, we recommend that the following object classes also be allowed to parent RPC object classes:

- country
- organization
- organizational Unit
- locality
- container

These object classes are included in the *Lightweight Directory Access Protocol: Standard and Pilot Attribute Definitions*, an ETF standard document, still in progress.

6.3.4.4 RPC Entry

The RPC Entry is the class from which all other RPC classes are derived.

```
( 1.2.840.113556.1.5.27
NAME 'rpcEntry'
SUP top
PARENT (rpcEntry $ rpcGroup $ rpcProfile $ rpcServer)
STRUCTURAL
MUST cn
     )
```

NOTE: The implementation treats **rpcEntry** as a structural, rather than an abstract, object class.

6.3.4.5 RPC Group

The **rpcGroup** object defines an RPC Group. The **cn** is the RDN component of the entry name provided by the user in the NSI API call that creates the group. The **rpcNsObjectID** attribute contains string UUIDs of objects added to the group entry by applications. These object IDs are not used by the NSI provider during lookup operations.

```
( 1.2.840.113556.1.5.80
NAME 'rpcGroup'
SUP rpcEntry
PARENT (rpcEntry $ rpcGroup $ rpcProfile $ rpcServer)
STRUCTURAL
MAY rpcNsGroup
MAY rpcNsObjectID
   )
```

The next-to-last code line, `MAY rpcNsGroup`, is changed here from the Open Group specification, which uses MUST. To make the attribute **rpcNsGroup** mandatory not only disallows the notion of empty groups, but also makes deleting a last member of a group impossible. Because both those operations must be supported by DCE, it is best to make the **rpcNsGroup** attribute optional instead of mandatory.

6.3.4.6 RPC Profile

RPC Profile entries are implemented by two object classes. The **rpcProfile** object class is a container used to gather profile elements into a single profile instance. The **cn** is the RDN component of the entry name provided by the user in the NSI API call that creates the profile.

```
( 1. 2 . 840.113556.1.5.82
NAME 'rpcProflle'
SUP rpcEntry
PARENT (rpcEntry $ rpcGroup $ rpcProflle $ rpcserver)
STRUCTURAL
      )
```

The **rpcProfileElement** object describes a single element in the profile. The entire profile is retrieved with a single-level LDAP search rooted at the parent **rpcProfile** container. The **cn** is a string UUID generated by the NSI provider when the **rpcProfileElement** instance is created.

```
( 1.2.840.113556.1.5.26
NAME 'rpcProfileElement'
SUP rpcEntry
PARENT rpcProfile
STRUCTURAL
MUST ( rpcNsPriorlty $ rpcNsProfileEnt an $ rpcNsInterfacQId )
MAY rpcNsAnnotation
       )
```

6.3.4.7 RPC Server

RPC Server entries are implemented by two object classes. The **rpcServer** object class is a container. It is used to gather **rpcServerElement** entries into a single server instance. The **cn** is the user-provided RDN component of the entry name in the NSI API call that creates the server entry.

```
( 1.2.840.113556.1.5.81
NAME 'rpcServer'
SUP rpcEntry
PARENT (rpcEntry $ rpcGroup $ rpcProfile $ rpcServer)
STRUCTURAL
MAY ( rpcNsObjectID $ rpcNsCodeSet )
        )
```

The **rpcServerElement** object describes a single interface in the server entry. The entire Server entry is retrieved with a single-level LDAP search rooted at the parent **rpcServer** container. The attributes of the **rpcServerElement** object allow for efficient searching using straightforward LDAP query expressions. The **cn** is a string UUID generated by the NSI provider when the **rpcServerElement** instance is created.

```
( 1.2.840.113556.1.5.73
NAME 'rpcServerElement'
SUP rpcEntry
PARENT rpcServer
STRUCTURAL
HOST ( rpcNsInterfaceID $ rpcNsBindings $ rpcNsTransferSyntax )
       )
```

6.3.4.8 Attribute Definitions

RPC Name Service implementations search on a well-known set of attributes. Implementations of this schema are advised for performance reasons to index the following attributes:

- rpcNsObjectID
- rpcNsInterfaceID

6.3.4.9 The rpcNsObjectID

A set of string UUIDs for objects (in the DCE RPC sense of objects):

```
( 1.2.840.113556.1.4.312
NAME 'rpcNsObjectID'
EQUALITY caseIgnoreListMatch
SYNTAX directoryString
USAGE userApplications
```

6.3.4.10 The rpcNsGroup

A set of DNs for RPC entries that are members of a given RPC group:

```
( 1.2.840.113556.1.4.11d
NAME 'rpcNsGroup'
EQUALITY distinguishedNameMatch
SUBSTRING distinguishedNameMatch
```

```
SYNTAX DN
D8AGE userApplications
)
```

6.3.4.11 The rpcNsPriority

An integer value indicating the priority of a given RPC profile element:

```
( 1.2.840.113556.1.d.117
NAME 'rpcNsPriority'
EQUALITY integerMatch
SYNTAX INTEGER
USAGE userApplicationa
)
```

6.3.4.12 The rpcNsProfileEntry

The DN of a single RPC entry that is a member of a given RPC profile:

```
( 1.2.840.113556.1.4.118
NAME 'rpcNsGroup'
EQUALITY distinguishedNameMatch
SUBSTRING distinguishedNameMatch
SYNTAX DN
SINGLE-VALUE
USAGE userApplications
)
```

6.3.4.13 The rpcNsInterfaceID

A string composed of the UUID for an interface exported by an RPC server and the interface major and minor version numbers in the form:

```
 string-UUID'',''major''.''minor
```

The BNF description of this item is:

```
 ( 1 2 840 113556 1 4 115
NAME 'rpcNsInterfaceID'
EQUALITY caseIgnoreMatch
SYNTAX directoryString
SINGLE-VALUE
USAGE userApplications
)
```

6.3.4.14 The rpcNsAnnotation

A string describing a given RPC Profile element:

```
( 1 2 840 113556 1 4 366
NAME 'rpcNsAnnotation'
EQUALITY caseIgnoreMatch
SUBSTRING caseIgnoreMatch
SYNTAX directoryString
SINGLE-VALUE
USAGE userApplications
)
```

6.3.4.15 The rpcNsCodeset

A set of strings identifying the character sets supported by a given RPC server:

```
( 1 2 840 113556 1 4 367
NAME 'rpcNsCodeset'
EQUALITY caseIgnoreListMatch
SYNTAX directoryString
USAGE userApplications
)
```

6.3.4.16 The rpcNsBindings

A set of binding strings for a given interface and transfer syntax, in the form:

```
ProtocolSequence'':''NetworkAddress''[]''
```

The BNF description of this item is:

```
( 1 2 840 113556 1 4 113
NAME 'rpcNsBindings'
EQUALITY caseIgnoreMatch
SYNTAX dlrectoryString
USAGE userApplications
)
```

6.3.4.17 The rpcNsTransferSyntax

A set of strings composed of the string UUID for a transfer syntax supported by an RPC server, and the transfer syntax major and minor version numbers in the form:

```
string-UUID'','major'' ''minor
```

The BNF description of this item is:

```
( 1 2 840 113556 1 4 314
NAME 'rpcNsTransferSyntax'
EQUALITY caseIgnoreLletMatch
SYNTAX directoryString
USAGE userApplications
)
```

6.3.5 Usage Model

Instantiating an **rpcGroup**, **rpcProfile**, or **rpcServer** object and any necessary child objects (e.g., **rpcServerElement** or **rpcProfileElement**) can create any RPC entry type. Searching is simplified because there is a well-known set of object classes and attributes for each entry type.

A group entry contains the **rpcNsGroup** attribute listing the entries in the group. Each group is a single object and can be retrieved in a single operation. An **rpcGroup** object can have **rpcNsObjectID** present—the list of object IDs, if present, is explicitly stored and retrieved by applications and not used by the NSI provider in locating **rpcNsGroup** objects.

A profile entry consists of an **rpcProfile** container with one or more **rpcProfileEntry** objects as children, one for each priority level defined. The complete profile is retrieved in a single operation by performing a single-level LDAP search for objects of class **rpcProfileElement** rooted at the **rpcProfile** entry.

A server entry consists of an **rpcServer** container with one or more **rpcServerElement** objects as children. The complete entry is retrieved in a single operation by performing a single-level LDAP search for objects of class **rpcServerElement** rooted at the **rpcProfile** entry. The NSI provider creates a new **rpcServerElement** entry when the interface and transfer syntax provided by the caller do not match an existing **rpcServerElement** in the named server entry. If a matching **rpcServerElement** exists, the NSI provider updates it with the string bindings provided by the caller.

This schema allows many discrete **rpcServerElement** objects to be stored in a given entry. This avoids a number of problems in trying to store multiple interfaces with their versions and transfer syntax in a single entry while providing convenient access and searching with LDAP. Indexing the Interface ID and Object UUIDs reduces the performance cost for retrieving multiple objects.

6.3.5.1 Relative Names

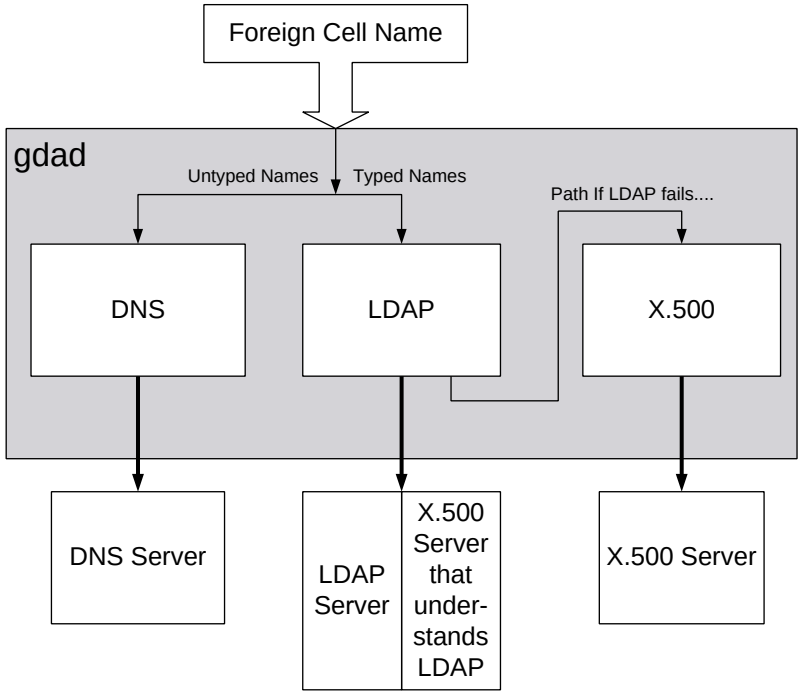
On the surface, CDS can be said to support cell-relative names; LDAP does not. DCE RPC allows names presented to the RPC NSI to be absolute or relative. An absolute name contains the full DN of the entry in question. A relative name is relative to the DCE cell where the name is stored. The full DN is the DN of the cell with the relative name appended. When using an LDAP directory to store RPC entries as defined by this schema, the implementation of relative names is implementation dependent, but should be consistent. A suggested approach is the creation of a container at the root of the namespace; for example, directly below the first instantiated object, such as Organization or organizational Unit, which forms the root for cell-relative names.

6.4 How GDA Works

LDAP support at the GDA level is achieved by adding an LDAP path for cross-cell information. Depending on the syntax of the cell name and if a specific path is enabled via command line options, GDA is now able to look up foreign cell information in either DNS, or LDAP, or X.500.

In cases when both LDAP and X.500 are enabled and the cell name is typed, GDA first resolves the name using LDAP, only if the typed name cannot be resolved, the X.500 path is invoked. *Figure 6-2* shows the various elements at work in the **gdad** environment.

Figure 6-2: Operation at gdad Level



6.4.1 Cell Naming

Cell names remain either as untyped names in DNS format or typed names in X.500 format. LDAP cell names are not supported.

If LDAP is enabled, GDA converts an X.500 typed cell name into LDAP syntax when sending a request to an LDAP directory service. The conversion routine can return unexpected results if special characters defined by either X.500 or LDAP are used in the cell name.

6.4.2 Security

GDA supports the minimum level of authentication. Authentication information may be supplied on the command line when **gdad** is started. However, these command line arguments can be viewed by most users and therefore a security problem can exist.

6.4.3 Registration Utility

The utility **ldap_addcell** is provided to register cell information in the LDAP directory service. The utility obtains and dynamically adds DCE cell information to the LDAP directory service. Authentication information must be provided on the command line.

CHAPTER 7

Managing Intercell Naming

7

7.1 Overview of Intercell Naming

To find names outside of the local cell, CDS clerks must have a way to locate directory servers in other cells. The Global Directory Agent (GDA) enables intercell communications by serving as a connection to other cells through the global naming environment. This chapter describes how the GDA works and how to manage it. The chapter also describes how to define the local cell in either of the global naming environments (DNS, X.500, or LDAP), where a step is necessary to make the local cell accessible to other cells.

NOTE: If the cell name is an X.500 formal name, then either GDS or an LDAP server may be used as the global name server.

7.2 How the Global Directory Agent Works

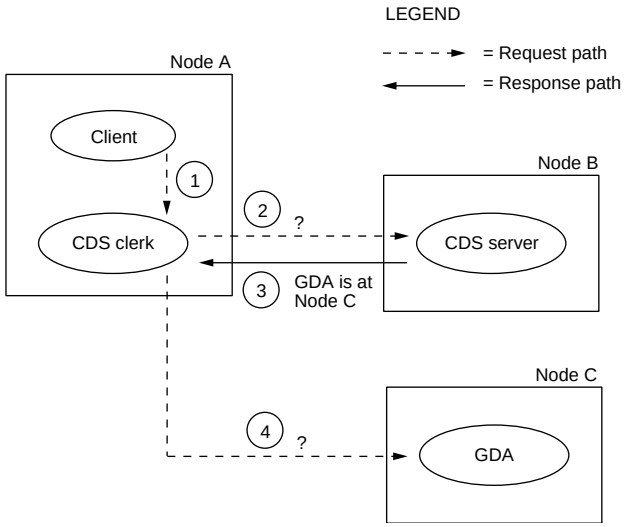
The GDA is an intermediary between CDS clerks in the local cell and CDS servers in other cells. A CDS clerk treats the GDA like any other name server, passing it name lookup requests. However, the GDA provides the clerk with only one specific service; it looks up a cell name in the X.500, LDAP, or DNS namespace and returns the results to the clerk. The clerk then uses those results to contact a CDS server in the foreign cell.

A GDA must exist inside any cell that wants to communicate with other cells. It can be on the same system as a CDS server, or it can exist independently on another system. You can configure more than one GDA in a cell for increased availability and reliability. Like a CDS server, a GDA is a principal and must authenticate itself to clerks.

CDS finds a GDA by reading address information that is stored in the **CDS_GDAPointers** attribute associated with the cell root directory. Whenever a GDA process starts, it creates a new entry or updates an existing entry in the **CDS_GDAPointers** attribute. The entry contains the address of the host on which the GDA is currently running. If multiple GDAs exist in a cell, they each create and maintain their own address information in the **CDS_GDAPointers** attribute.

When a CDS server receives a request for a name that is not in the local cell, the server examines the **CDS_GDAPointers** attribute of the cell root directory to find the location of one or more GDAs. The next figure shows how a CDS clerk and CDS server interact to find a GDA.

Figure 7-1: How the CDS Clerk Finds a GDA



The following steps summarize the GDA search that is illustrated in the preceding figure:

1 On Node A, a client application passes a global name, beginning with the **/...** prefix, to the CDS clerk.

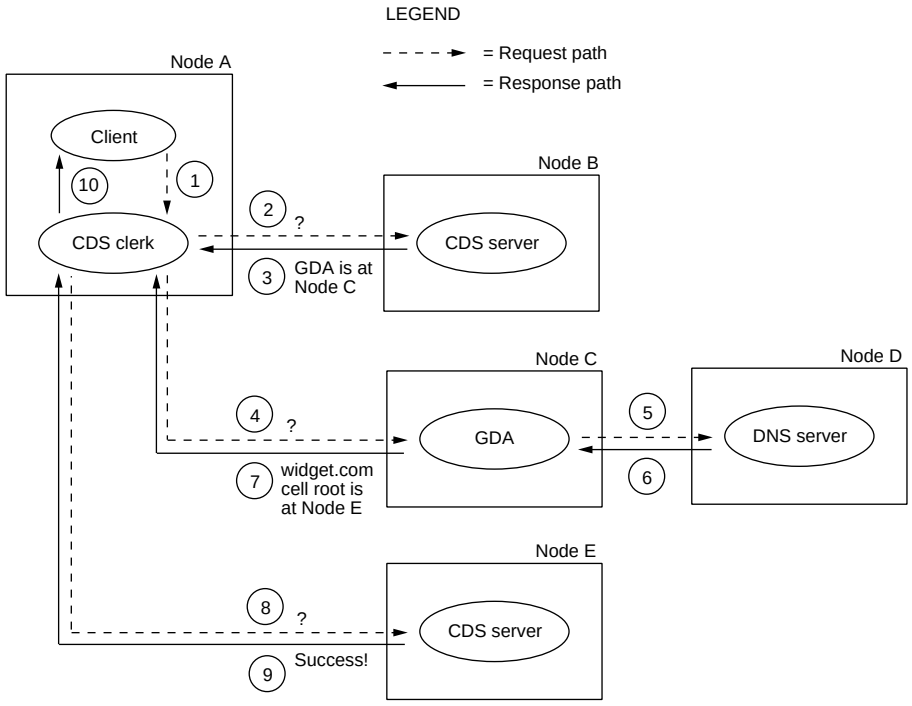
2 The clerk passes the lookup request to a CDS server that it knows about on Node B.

3 The server's clearinghouse contains a replica of the cell root directory, so the server reads the **CDS_GDAPointers** attribute and returns the address of Node C, where a GDA is running.

4 The clerk passes the lookup request to the GDA.

The next figure shows how CDS and a GDA interact to find a name in a foreign cell that is defined in DNS. Suppose the name is **/.../widget.com/printsrv1**, which represents a print server in the foreign cell.

Figure 7-2: How the GDA Helps CDS Finds a Name



The following steps summarize the name search that is illustrated in the preceding figure:

1 The client application passes the name **/.../widget.com/printsrv1** to the CDS clerk.

2 The clerk passes a lookup request to a CDS server that it knows about on Node B.

3 The server's clearinghouse contains a replica of the cell root directory, so the server looks up the **CDS_GDAPointers** attribute and returns the address of Node C, where a GDA is running.

4 The clerk passes the lookup request to the GDA.

5 The GDA recognizes that the name is a DNS-style name, so it assumes that the second component is a cell name that is defined in DNS. It passes that portion of the name (**widget.com**) to DNS. For simplicity, the figure shows only one DNS server; more than one DNS server can actually be involved in resolving a global cell name.

NOTE: Although this example concerns the lookup of a DNS-style name, the sequence and execution of operations is nearly identical for an X.500 name or a hierarchical cell name. If the GDA recognizes that a name is an X.500-style name, it passes the name to either an LDAP client (via LDAP APIs) or a GDS client (via XDS/XOM APIs) rather than to a DNS server. The LDAP client or GDS client then communicates with the appropriate server to obtain the cell bindings (the same information as would be obtained from a DNS server). If the GDA recognizes that a name is a hierarchical cell name, it passes it to the CDS server of the topmost cell in the hierarchy, which is registered in one of the global namespaces. The CDS server in this cell walks down the cell hierarchy to locate the name.

6 DNS looks up and returns to the GDA information that is associated with the **widget.com** cell entry. The information includes the addresses of servers that maintain replicas of the root directory of the **/.../widget.com** cell namespace.

7 The GDA passes the information about the foreign cell to the clerk.

8 The clerk contacts the CDS server on Node E in the foreign cell, passing it a lookup request.

9 The Node E server's clearinghouse contains a replica of the root directory, so the server looks up the entry for **printsrv1** in the root and passes the requested information to the clerk on Node A. For simplicity, this example shows the clerk contacting only one server in the foreign cell. While resolving a full name, the clerk may actually receive referrals to several servers in the foreign cell.

10 The clerk passes the information to the client application that requested it.

Note that both of the previous examples represent initial lookups. The CDS clerk caches the locations of GDAs once it discovers them. The clerk also caches the addresses of servers in foreign cells that it learns about, enabling it to contact the foreign servers directly on subsequent requests for names in the same cell.

Note also that a GDA knows its own cell name and can therefore avoid contacting a global directory service to look up names in its own cell. Furthermore, the GDA can recognize whether a cell name conforms to the X.500 or DNS naming syntax, and it uses that knowledge to route a lookup request to the appropriate global directory service. If the cell name conforms to the X.500 naming syntax, the GDA will first send the request to the LDAP client and then to the GDS client if it is not resolved by the LDAP client/server.

7.3 Managing the Global Directory Agent

Use the DCE configuration program to configure the GDA; the GDA requires little management once it is configured. (See the *OSF DCE Administration Guide—Introduction* for details on configuring the GDA.)

The GDA is typically started and stopped automatically by scripts that execute as part of normal system startup and shutdown procedures. Sometimes, however, you may want to use commands to stop and restart a GDA. Once you have configured GDA with the DCE configuration program, you can use these steps to start and stop GDA.

The GDA runs as a process called **gdad**. To start the **gdad** process, follow these steps:

1 Make sure that a CDS server is already running somewhere within the cell.

2 Log into the system as superuser (**root**).

3 Enter the following command to see if the **dced** process is already running:

 # ps

If the **dced** process appears on the list of active processes, proceed to step 5. If the **dced** process does not appear on the list of active processes, enter the following command to start the process:

dced

4 Enter the following command to start the **cdsadv** process:

cdsadv

5 Enter the following command to start the **gdad** process:

gdad

NOTE: See the *OSF DCE Administration Guide—Introduction* for the parameters required if **gdad** is to use LDAP to obtain cell bindings.

To stop the GDA, enter the following command, where *pid* is the process identifier of the gdad process:

kill *pid*

7.4 Enabling Other Cells to Find Your Cell

The GDA is the mechanism that allows CDS clerks in your local cell to find other cells. To make your cell accessible to others, you must create an entry for it in one of the currently supported global naming environments. Before you do this, obtain a unique cell name from the appropriate naming authority. (See the *OSF DCE Administration Guide—Introduction* for details.)

After you configure a cell, name it, and initialize the cell namespace, you can use the **dcecp directory show** command to obtain the data you need to create or modify the cell entry in an X.500, LDAP, or DNS server. You can use the **ldap_ addcell** command to add the appropriate information for the cell to an LDAP server. The data in a cell entry is what the GDA passes to CDS after looking up a cell name. CDS, in turn, uses the information to contact servers in the cell. The following subsections describe how to define and maintain a cell entry in an X.500 server (GDS), an LDAP server, or DNS server. These sections assume a basic familiarity with X.500 and DNS; for details, see the appropriate documentation for each global name service.

You can also define and maintain a cell entry in the CDS namespace of another cell. This type of definition exists in a hierarchical cell configuration.

7.4.1 Defining a Cell in the Domain Name System

Names in DNS are associated with one or more data structures called *resource records*. The resource records are stored in a data file whose name and location are implementation specific. To create a cell entry, you must edit the data file and create two resource records for each CDS server that maintains a replica of the cell namespace root.

The first resource record, whose type can be AFSDB or MX, contains the host name of the system where the CDS server resides. You can use MX as an alternative to using AFSDB. The second record, of type TXT, contains the following information about the replica of the root directory that the server maintains:

- The UUID of the cell namespace, in hexadecimal notation
- The type of the replica (master or read-only)
- The global CDS name of the clearinghouse where the replica resides
- The UUID of the clearinghouse, in hexadecimal notation
- The DNS name of the host where the clearinghouse resides

The following example shows a set of AFSDB resource records for a cell that is named **cs.tech.edu**, in which two replicas of the root directory exist. Note that only the first resource record contains the cell name; the second, third, and fourth records are assumed to be associated with the same cell because they do not contain a cell name. The **TTL** heading stands for time-to-live, which is a value, in seconds, after which the data is no longer considered valid in a DNS cache. (The value shown specifies a default value of 1 week.) The **IN** class indicates that the protocol is Internet, and the subtype of **2** indicates that a name server exists on the host named in the record.

```
;First Replica:
;Name            TTL       Class     Type     Subtype   Host
cs.tech.edu      604800    IN        AFSDB     2        fox.cs.tech.edu
;Name            TTL       Class     Type     Rdata
                 604800    IN        TXT      (1        ;version
                 fd3328c4-2a4b-11ca-af85-09002b1c89bb   ;ns uuid
                 Master                                 ;Replica1 type
                 /.../cs.tech.edu/cs1_ch                ;ch name
                 fd3328c5-2a4b-11ca-af85-09002b1c89bb   ;ch uuid
                 fox.cs.tech.edu)                       ;host
;Second Replica:
                  604800    IN        AFSDB     2        rox.cs.tech.edu
                 604800    IN        TXT      (1        ;version
                 fd3328c4-2a4b-11ca-af85-09002b1c89bb   ;ns uuid
                 Read-only                              ;Replica2 type
                 /.../cs.tech.edu/cs2_ch                ;ch name
                 fd3429c4-2a4b-11ca-af87-09002b1c89bb   ;ch uuid
                 rox.cs.tech.edu)
;host
```

You can use MX as an alternative to using AFSDB. The following example shows a set of MX resource records for the same cell, **cs.tech.edu**, in which two replicas of the root directory exist.

```
;First Replica:
;Name            TTL       Class     Type    Preference   Exchange
cs.tech.edu.     604800    IN        MX      1            fox.cs.tech.edu.
;Name            TTL       Class     Type    Rdata
                 604800    IN        TXT     (1           ;version
                 fd3328c4-2a4b-11ca-af85-09002b1c89bb     ;ns uuid
                 Master                                   ;Replica1 type
                 /.../cs.tech.edu/cs1_ch                  ;ch name
                 fd3328c5-2a4b-11ca-af85-09002b1c89bb     ;ch uuid
                 fox.cs.tech.edu)                         ;host
;Second Replica:
                 604800    IN        MX      2
rox.cs.tech.edu.
                 604800    IN        TXT     (1           ;version
                 fd3328c4-2a4b-11ca-af85-09002b1c89bb     ;ns uuid
                 Read-only                                ;Replica2 type
                 /.../cs.tech.edu/cs2_ch                  ;ch name
                 fd3429c4-2a4b-11ca-af87-09002b1c89bb     ;ch uuid
                 rox.cs.tech.edu)
;host
```

After you configure a cell, you can use the **dcecp directory show** command to display the information that is required to construct resource records like those shown in the previous example. The following is an example **directory show** command that displays output for a cell named **/.../cs.tech.edu**.

```
dcecp> directory show /.../cs.tech.edu
```

To create a new resource record in the DNS namespace, use the information from the **directory show** command and place the properly formatted data into the DNS data file.

7.4.2 Defining a Cell in the Global Directory Service

In GDS, cell information is contained in two attributes: **CDS-Cell** and **CDS-Replica**. You can cause an existing GDS name to become a cell entry by adding these two attributes to the name. If the name you want to use for the cell does not yet exist, you must create it and then add the attributes. The GDS administration program uses numbered screens called *masks* to accept user input. Use the object administration masks to create a cell entry. (See the *OSF DCE GDS Administration Guide and Reference* for details.)

After you configure a cell, you can use the **dcecp directory show** command to obtain the data that you need to supply when you are creating the **CDS-Cell** and **CDS-Replica** attributes. The following is an example directory show command and the resulting GDS-formatted output for a cell that is named **/.../C=US/O=ABC/OU=Sales**:

```
dcecp> directory show /.../C=US/O=ABC/OU=Sales
{RPC_ClassVersion {01 00}}
{CDS_CTS 1996-04-18-20:11:02.385764100/08-00-09-85-01-22}
{CDS_UTS 1996-08-01-18:01:37.408282100/08-00-09-85-01-22}
{CDS_ObjectUUID 68f0755c-9956-11cf-9da3-080009850122}
{CDSReplicas
  {{CH_UUID 59eb61fc-9956-11cf-9da3-080009850122}
   {CH_Name /.../c=us/o=abc/ou=sales/dcegecko_ch}
   {Replica_Type Master}
   {Tower {ncadg_ip_udp 15.22.50.148}}
```

```
   {Tower {ncacn_ip_tcp 15.22.50.148}}}}
{CDS_AllUpTo 1996-08-01-14:39:36.404042100/08-00-09-85-01-22}
{CDS_Convergence medium}
{CDS_ParentPointer
  {{Parent_UUID 5a824f54-9956-11cf-9da3-080009850122}
   {Timeout
    {expiration 1996-08-02-14:01:36.251}
    {extension +1-00:00:00.000I0.000}}
   {myname /.../c=us/o=abc/subsys}}}
{CDS_DirectoryVersion 3.0}
{CDSReplicaState on}
{CDS_ReplicaType Master}
{CDS_LastSkulk 1996-08-01-14:39:36.404042100/08-00-09-85-01-22}
{CDS_LastUpdate 1996-08-01-18:01:37.408282100/08-00-09-85-01-22}
{CDS_Epoch 68fdf042-9956-11cf-9da3-080009850122}
{CDS_ReplicaVersion 3.0}
dcecp>
```

To create a new resource record in GDS, use the information from the **directory show** command to fill in the fields of Mask 21 (**CDS-Cell**) and Mask 22 (**CDS-Replica**) in the GDS administration program.

7.4.3 Defining a Cell in an LDAP Server

The **ldap_addcell** utility obtains and dynamically adds DCE cell information to an LDAP server. The **ldap_addcell** command must be run with root authority. The **ldap_addcell** command can:

- Create a new directory object with cell bindings.
- Modify an existing directory object to add the cell bindings.
- Change the values of the cell bindings in a directory object that already exists.
- Delete the cell bindings from a directory object that already exists.

The cell bindings that are added or retrieved from a directory object have the same format used for an X.500 server (GDS) and are stored in 2 attributes:

- CDSCELL
- CDSREPLICAS

Authentication information such as **userid** and **password** are part of the **ldap_addcell** utility invocation, because it writes to the directory service. The DCE cell information stored in the directory service is the same whether it was written using the X.500 registration utility or the **ldap_addcell** registration utility.

The **ldap_addcell** command has the following syntax:

ldap_addcell -h *ldap_server* **-a** *authentication_DN* **-p** *password* [**-o** *object_class,object_class...*]|[**-d**]

where:

-h *ldap_server*	The name of the LDAP server targeted to hold the cell binding.
-a *authentication_DN*	The distinguished name (DN) specified in LDAP name syntax that will be authenticated and used to add cell binding.
-p *password*	The password that is used to authenticate the distinguished name (DN).
-o *object_class*	Value(s) of the attribute object_class for the entry (the registration) being created or modified. Note that, if you are listing more than one *object_class* value, you must separate them with commas.
-d	Deletes the DCE cell information attributes from the entry in the directory. It does not remove the entire directory entry.

The command must be run with root authority and prints a message to `stderr`.

The following **ldap_addcell** examples assume the following:

- *mymachine.mycity.mycompany.com* is the LDAP server machine name.
- `gdatest` is a user that has write access to the LDAP server.
- `gdatest` is also the password of the user gdatest.
- An `organizationalUnit` is allowed to contain the auxiliary object, `dceCellInfo`.
- The LDAP server does schema checking.

This example shows the normal creation of the cell bindings in the LDAP server.

```
ldap_addcell -h mymachine.mycity.mycompany.com -a
"cn=gdatest,ou=houston,o=compaq,c=us" -p "gdatest"        -o
organizationalUnit,dceCellInfo
```

This example shows the deletion of the **CDSCELL** and **CDSREPLICAS** attributes.

```
ldap_addcell -h mymachine.mycity.mycompany.com -a
"cn=gdatest,ou=houston,o=compaq,c=us" -p "gdatest" -d
```

This example shows the changing of the CDSCELL and CDSREPLICAS attributes in an object that already exists.

```
ldap_addcell -h mymachine.mycity.mycompany.com -a
"cn=gdatest,ou=houston,o=compaq,c=us" -p "gdatest
```

Most parameters of the **ldap_addcell** command have a corresponding environment variable which is used when the corresponding parameter is not present on the **ldap_addcell** command invocation. *Table 7-1* lists environment variables.

Table 7-1: ldap_addcell Parameters and Environment Variables

ldap_addcell Parameter	Environment Variable
-h	LDAP_SERVER
-a	LDAP_AUTH_DN

Table 7-1: ldap_addcell Parameters and Environment Variables

ldap_addcell Parameter	Environment Variable
-p	LDAP_AUTH_DN_PW
-o	LDAP_OBJECT_CLASS

NOTE: The **-d** parameter does not have a corresponding environment variable.

If the cell entry is already registered, the **CDSCELL** and **CDSREPLICAS** attributes are replaced with new values for this cell unless the **-d** parameter is specified.

CHAPTER 8

DCE Distributed File Service

8

8.1 Variation from OSF DFS

Gradient DCE for Tru64 UNIX includes DCE DFS from OSF DCE Release 1.2.2. This release does not contain any enhancements for DFS beyond those that are part of OSF DFS. However, there are the following areas of difference:

- The Episode file system is not supported.
- DFS in Gradient DCE for Tru64 UNIX does not include enhanced DFS features such as fileset cloning.
- DFS in Gradient DCE for Tru64 UNIX allows the use of Tru64 UNIX ACLs for authorization purposes.
- DFS in Gradient DCE for Tru64 UNIX relies on Tru64 UNIX built-in file system backup rather than using the backup facility provided with OSF DFS.

For information on how to configure DFS, see the *Gradient DFS for Tru64 UNIX Configuration Guide*.

The last section in this chapter identifies solutions to some common problems you might encounter using DFS.

8.2 Using Tru64 UNIX ACLs

Tru64 UNIX supports the use of generic ACLs on its two supported filesystems (UFS and AdvFs). The ACLs follow the POSIX model, providing a sequence of ACL entries, each consisting of a tag (type), an identifier for entries whose type requires it, and a set of permission bits.

Table 8-1: Tru64 UNIX ACLs

Tag	Identifier	Permission Bits
user	uid	rxw
group	gid	rxw
user_obj		rxw
group_obj		rxw
other_obj		rxw

ACL entries tagged as **user** or **group** identify persons or groups that might attempt to perform some action on the directory or file. The Identifier is a user id (uid) for **user** tags or a group identifier (gid) for **group** tags. ACL entries tagged as **user_obj**, **group_obj**, and **other_obj** do not use identifiers because these are implicit in the metadata of the directory or file. (See Note below.) The permissions are the standard UNIX read (**r**), write (**w**), and execute (**x**) permissions.

NOTE: Because DFS in Gradient DCE for Tru64 UNIX maps uids and gids to specific users and groups, password files must be synchronized with the DCE Security registry. Enabling Security Integration Architecture (SIA) offers one way to synchronize uid and gid information with the DCE cell registry.

Default ACLs for containers and objects are created following the same method as in the standard DCE DFS implementation.

8.2.1 Tru64 UNIX ACL Limitations

Tru64 UNIX ACLs lack the following functionality that is available with generic DCE ACLs:

- A set of “foreign” tags supporting users, groups, and objects from foreign cells.
- A set of “delegation” tags supporting delegation from users, groups, and objects in the local cell and in foreign cells.
- An unauthenticated mask controlling access for unauthenticated users.
- A cell name included in ACL identifiers which is used for foreign cell user authentication.
- A wider set of permission bits:
  - (**c**) control
  - (**i**) insert
  - (**d**) delete

An additional limitation of Tru64 UNIX ACLs is that the ACL identifiers are uids or gids instead of full DCE UUIDs.

Gradient DCE for Tru64 UNIX handles these ACL limitations by providing appropriate responses to administrative or user actions that involve Tru64 UNIX ACLs. People or programs that use or administer DFS proceed as normal DCE clients. A transparent translation layer in DCE DFS intercepts and deals with ACL operations.

8.2.2 DCE Responses to Tru64 UNIX ACL Operations

Due to the limitations of Tru64 UNIX ACLs, some operations involving ACLs behave differently or return an error. Specific responses to Tru64 UNIX ACL operations depend on whether the operation is unsupported, totally supported, or partially supported.

Unsupported operations such as adding an entry for **foreign_user**, or **group_delegate** return an error.

Totally supported operations such as a user in the local cell requesting write access to a file behave in the standard manner.

Some operations are partially supported. Tru64 UNIX provides appropriate responses to certain operations even though the features for their support is lacking from the Tru64 UNIX ACLs. For example, a user attempts to delete a file from DFS. Normally, DFS requires the **d** (delete) permission but Tru64 UNIX performs the delete operation if the user has write permission on the file.

8.2.3 Mapping between DCE ACLs and Tru64 UNIX ACLs

The mapping is done by a translation layer between DFS and the underlying physical file system at the server. In other words, none of this work has any bearing on the client portion of DFS.

- There is no space for a home cell uuid, so the server assigns the UUID of the cell that it belongs to as the home cell UUID of any ACL that it deals with.
- No “foreign” ACL entries are possible. The client can submit them, but the cell UUID is dropped before the mapping to a uid or gid is done (the mapping will fail in this case, since the foreign user or group UUID will not be found in the registry of this cell).
- The mapping between principal or group UUIDs on one hand and uid/gids on the other is done by querying the registry of the cell to which the file server belongs. It is assumed that the password files are synchronized with the registry or a scheme like SIA is used.
- The permission bits need to be mapped according to *Table 8-2*.

Table 8-2: Mapping Permission Bits

Tru64 UNIX ACL Bits	DCE ACL Bits	
	file	directory
r	r	r
w	cw	cwid
x	x	x

- DFS simulates a **mask_obj** tag to satisfy operations that require its presence. However, the simulated **mask_obj** does not mask any permissions (its permissions are **rwxcid**).
- The **initial_container** and **initial_object** ACLs behave normally.

8.2.4 Disabling ACL Operations

You can disable the ACL support in the DFS server by setting a kernel global variable using the dbx debugger. After a new kernel that includes DFS support has been built, specify the following:

```
cd /usr/sys/conf
dbx -k vmunix
patch dfs_acls_enabled = 0
quit
```

where *conf* is the name of the configuration you chose when executing doconfig. After disabling ACL, any remote ACL operations on DFS files return ENOTTY errors.

8.3 NFS-DFS Secure Gateway Server Administration

The NFS-DFS Secure Gateway server does not support the **dfs_login** and **dfs_logout** programs. For authenticated access to DFS, users of DCE-unaware NFS clients must authenticate to DCE from the Gateway Server machine using a **dfsgw add** operation. Refer to the *OSF DCE DFS Administration Guide and Reference* for information about authenticating from a Gateway Server machine.

8.4 DFS Backup

DFS in Gradient DCE for Tru64 UNIX relies on Tru64 UNIX built-in file system backup rather than using the backup facility included with OSF DFS. Refer to your Tru64 UNIX documentation for instructions on using the Tru64 UNIX file system backup facility.

8.5 Solutions to Common Problems with DCE DFS

Here are solutions to a few common problems that you may encounter with DCE DFS.

8.5.1 Running Commands Requiring the setuid Feature

Commands that use the **setuid** feature (for example, the **ps** command) do not execute properly if used from the DFS namespace. Before running the commands, you must enable the setuid functionality on a per fileset basis by issuing the **cm setsetuid** command. Issue this command on each machine that needs to use these **setuid** commands after DFS has started, that is, after the system is in multiuser mode. See **cm setsetuid(8dfs)** in the *OSF DCE DFS Administration Guide and Reference* for more information.

8.5.2 Running cron Jobs with DCE Credentials

It is often necessary to run jobs asynchronously with DCE credentials. For example, you might run a job after hours that requires access to DFS. One way to have a job running under **cron**(1) or **at**(1) acquire DCE credentials is

by using the **-k** option of the **dce_login** command. This option allows **dce_login** to acquire credentials by reading a key from a keytab file, rather than by getting a password interactively. Using the **-k** option along with the **-e** option, which allows an executable command to be specified on the command line, accomplishes the desired effect.

The solution consists of two parts:

- First, decide on a principal with whose credentials the **cron** job should run. (Create a DCE user for this, if one does not exist already.) In the following example, the principal is designated with the placeholder *princ*. Then, as cell_admin, create a keytab file with a command similar to the following:

  ```
  dcecp -c keytab create princ.keytab \
              -storage /path/name/of/keytab \
              -data {princ plain 1 password}
  ```

  Where the *password* is the same password that was specified when the *princ* account was created in DCE. You may need the **-noprivacy** option if you do not have the privacy kit installed on the machine. The keytab file is created with **root** as the owner and 600 permissions. The ownership of the file has to be changed to the UNIX identity of the executor of the **cron** job.

- Next, you can add a line similar to the following to a crontab file to have **cron** run a script with the credentials of principal *princ*:

  ```
  5 20 * * 1-5 dce_login princ -k /path/name/of/keytab \ -e /path/name/of/
  script
  ```

  to run the indicated script with the credentials of *princ* at 8:05 p.m., Monday through Friday.

You can verify that the first step above worked by issuing the following command:

```
dce_login princ -k /path/name/of/keytab -e klist
```

and making sure that the principal listed is indeed *princ*.

CHAPTER 9

Compiling and Linking Applications

9

9.1 Overview of the Command Format

This chapter describes the command format for compiling and linking DCE applications on Tru64 UNIX systems.

Note that you can use either the **cc** compiler or the **c89** compiler.

Every module of a DCE application program begins with the included header file **pthread.h**, as shown in the following example:

```
#include <pthread.h>
```

If **pthread.h** is not first in the include list for each module, the compiler can generate warning or error messages about the prototypes for these routines. In particular, it is best to precede those files containing call declarations for which there are jacket routines (such as **stdio.h**).

When linking a DCE application, you must use the **-threads** option.

Tru64 UNIX Version 4.0*x* supports the updated pthread standard, POSIX 1003.1c (D10), in addition to a backward-compatibility mode for the previous draft POSIX 1003.4a (D4).

DCE is built using the POSIX 1003.4a interfaces and the DCE documentation on pthreads corresponds to the 1003.4a standard. Until all DCE vendors support the new standard, we recommend that you continue to build your applications using POSIX 1003.4a interfaces.

To use interfaces defined in POSIX 1003.4a, compile all modules using **-DPTRHEAD_USE_D4** and link the application using the **-threads** option in the loader.

The following command format is an example of how to compile and link:

```
% cc -o myprog myprog.c -DPTHREAD_USE_D4 -threads
```

For more information on pthreads for Tru64 UNIX refer to the *Guide to DECthreads* and to the reference pages on the **ld** command.

Note that the **cc** and **c89** compilers do not define **__STDC__** by default. If you want to include ANSI C function prototypes in your application, you must specify the **-std1** option on the C compiler command line.

For complete information on compiling and linking applications, refer to the *OSF DCE Application Development Guide*.

CHAPTER 10

RPC, IDL, ACF, and IDL Compiler Enhancements

10

10.1 Overview of Enhancements

This chapter describes enhancements to RPCs, IDL, and the ACF language:

- Localrpc
- DTSD timing
- Environment variables
- Automatic binding can use host's profile
- Enumeration enhancements
- The **client_memory** ACF attribute provides more memory control

10.2 Local RPC Protocol Sequence

Gradient DCE for Tru64 UNIX now supports a new protocol sequence (protseq) in addition to TCP, UDP, DECnet, and OSI (ncacn_ip_tcp, ncadg_ip_udp, ncacn_dnet_nsp, and ncacn_osi_dna protocol sequence strings, respectively). The new protocol sequence is implemented with UNIX domain sockets and can be used only by clients and servers that are on the same node. The protocol sequence name is *localrpc*.

By using UNIX domain sockets, the IP layer can be bypassed, giving performance gains that vary with the nature of the RPC traffic.

This is not a "transparent" implementation that switches the user automatically to UNIX domain sockets when appropriate; rather, the user must explicitly use the *localrpc* protocol sequence in either a well-known endpoint in the IDL file, or as called out by one of the family of rpc_server_use_protseq*() functions (where * can be any characters) wherever a protocol sequence string can be used. String bindings can also be used to pass localrpc binding information from server to client.

10.2.1 Using localrpc with well-known endpoints

Within the IDL file, the user might have previously had an endpoint section as follows:

```
endpoint("ncadg_ip_udp:[2001]", "ncacn_ip_tcp:[2001]",
    "ncacn_dnet_nsp:[my_app_server]", "ncacn_osi_dna:[2001]")
```

Now this section can be expanded to:

```
endpoint("localrpc:[/tmp/my_app_server]",
   "ncadg_ip_udp:[2001]", "ncacn_ip_tcp:[2001]",
           "ncacn_dnet_nsp:[my_app_server]", "ncacn_osi_dna:[2001]")
```

If the corresponding server code calls rpc_server_use_all_protseqs_if(), a UNIX domain socket is created at /tmp/my_app_server in addition to using the rest of the protseqs specified in the endpoint section. A client residing on the same node as the server can then connect to the server using this socket to gain some performance advantage.

10.2.2 Affected RPC API calls

The following RPC API calls have been affected in some way by localrpc, or in some cases significantly not affected.

rpc_server_use_protseq()

This function can now be handed “localrpc” for the protseq parameter and a socket in the form of /tmp/LOCAL_RPC_42670001 will be dynamically formed, where “4267” is the hex form of the server pid, and the “0001” is a counter to ensure a unique name. The hex version of the pid is used in cleaning up unused sockets by dced.

rpc_server_use_protseq_ep()

This function can now be handed “localrpc” for the protseq parameter and a pathname for the endpoint. The socket is created at the given endpoint.

rpc_server_use_protseq_if()

This function can now be handed “localrpc” for the protseq parameter and the endpoint from the interface specification will be used.

rpc_server_use_all_protseqs()

This function does not use localrpc in the list of valid protseqs. This is to prevent breaking existing programs that randomly pick a binding and attempt endpoint operations using it. Endpoint database operations do not accept localrpc endpoints.

rpc_server_use_all_protseqs_if()

This function will now create a localrpc binding if it has been specified (along with any other protseqs) in the interface specification (IDL file).

rpc_ep_register()

This function will not put any localrpc binding handles into the endpoint database. If the entire binding vector consists only of localrpc binding handles, then the status rpc_s_no_bindings is returned. Otherwise, localrpc bindings are skipped over and other bindings are inserted.

rpc_ep_register_no_replace()

See rpc_ep_register() for identical restrictions.

rpc_string_binding_compose()

This routine will now accept localrpc in addition to the other protocol sequences already supported. When composing a string binding that has localrpc as its protocol sequence, it is only necessary to provide a NULL for the hostname as the parameter placeholder. The hostname, hostname IP, or “localhost” would also be acceptable.

rpc_binding_from_string_binding()

For localrpc string bindings, the hostname can be omitted. See the string binding style in the next call, rpc_binding_to_string_binding().

rpc_binding_to_string_binding()

For localrpc bindings, the hostname is not used in the string binding returned from this routine. Instead of localrpc:hostname:[/tmp/sockname], it simply uses: localrpc:[/tmp/sockname]

rpc_network_inq_protseqs()

This routine does *not* return localrpc in the list of supported protseqs to maintain compatibility with existing programs. Some existing programs call this routine, pick a binding, and use only that one binding in rpc_server_use_protseq() and rpc_ep_register() calls. The rpc_ep_register() call with only a localrpc binding would fail.

rpc_network_is_protseq_valid()

This routine will now return true for “localrpc,” as well as the previously supported protseqs.

10.2.3 Suppressing localrpc (or any other protseq)

Particular protseqs can be suppressed from RPC's consideration by listing only the desired protseqs in the RPC_SUPPORTED_PROTSEQS environment variable, for example:

```
setenv RPC_SUPPORTED_PROTSEQS ncacn_ip_tcp:ncadg_ip_udp
```

This example effectively suppresses DECnet, OSI, and Local-RPC protocol sequences. (Note: DECnet/OSI protocols will not be used if DECnet/OSI has not been configured.)

10.2.4 Permissions of localrpc Socket

The UNIX permission used on the socket created will normally follow from the user's current umask value and the permissions of the directory where the socket is created. This can be tuned by using a “permission network option” in the endpoint. For example, if the interface specification (IDL file) contained:

```
endpoint("localrpc:[/tmp/my_app_server,perm=666]",
   "ncadg_ip_udp:[2001]", "ncacn_ip_tcp:[2001]",
         "ncacn_dnet_nsp:[my_app_server]", "ncacn_osi_dna:[2001]")
```

The socket permission will be set to read and write for all users regardless of the current umask value. The network option can also be used when directly calling rpc_server_use_protseq_ep().

10.2.5 Added dced Support

Any localrpc sockets created (dynamically) in the form of

`/tmp/LOCAL_RPC_%4x%4d (e.g. /tmp/LOCAL_RPC_42670001)`

by using the rpc_server_use_protseq() function will be garbage-collected by dced some time after the server process has gone away. The reaper thread that accomplishes this will only be activated 3 times a day to keep the additional overhead very low.

10.2.6 Compatibility Issues

Localrpc endpoints will not be inserted into the Endpoint Database so that remote clients will not waste time attempting to connect to them. (See the rpc_ep_register() and rpc_ep_register_no_replace() functions in the above API section.)

Localrpc entries in the endpoint section of the interface specification are ignored by platforms that don't support this protocol sequence.

10.3 DTSD Timing and Timeout Changes

Within RPC, the API is extended with the effect of distributing timing signals from a different source and reducing the default TCP timeout period from 2 hours to 10 minutes. Previously, dtsd listened for the DECnet time service (DECdts) synchronization messages on data link interface (DLI). Now, dtsd defaults to RPC only.

To overturn the new default value (RPC only) and return to the former way of accepting timing messages:

```
% dtsd  -m
```

The dtsd command invokes the DTS daemon (server or clerk process). This command is usually executed as part of the overall DCE startup script, dcesetup.

You can enter the command manually under the following conditions:

- If a DTS daemon fails to start automatically upon reboot
- If you want to restart a daemon that you shut down to perform a backup or do diagnostic work

In normal rebooting, the **rc.dce** script automatically provides arguments appropriate to the choice of configuration options.

If **dtsd** is started with no arguments (other than **-d** for debugging and **-w** for serviceability determinations), then the server must be started with **dcecp**. The following example configures a local server:

```
dcecp> dts configure -notglobal

dcecp> dts activate
```

DTS runs as the host machine principal, which is usually **root**.

Use **dtsd** interactively only when troubleshooting; use the **/sbin/rc3.d/S66dce** script to start the DTS daemon. On some systems the superuser is associated with the machine principal.

10.3.1 Affected RPC API Call

The following RPC API calls has been affected by setting the default TCP timeout from (kernel tunable) two hours to (DCE) ten minutes:

rpc_mgmt_set_com_timeout()
rpc_mgmt_set_server_com_timeout()

Previously, these functions recognized only two timeout values for TCP connections. The two values were 0 or 10: rpc_c_binding_min_timeout(0) or rpc_c-binding_infinite_timeout(10). Those same calls recognized all integer values from 0 to 10 for UDP connections, however.

Currently, all values from 0 to 10 affect the timeout properties of each call without regard for the protocol selected. A value of 10 corresponds to two hours with each lower integer corresponding to a smaller period.

10.4 Using Environment Variables to Restrict Network Interfaces and Addresses

This section describes two environment variables that are useful controls for cluster environments and for systems with more than one network interface:

- RPC_UNSUPPORTED_NETIFS removes device(s) from RPC consideration.
- RPC_SUPPORTED_NETADDRS specifies network resources for RPC consideration.

```
% setenv RPC_UNSUPPORTED_NETIFS te1
```

Remove the device named “te1” from RPC consideration.

```
% setenv RPC_UNSUPPORTED_NETIFS te1:te2
```

Remove the two devices named “te1”and “te2” from RPC consideration.

NOTE: To list more than one device, use a colon-separated list.

```
% setenv RPC_SUPPORTED_NETADDRS 16.20.16.144
```

Of all network addresses that are available, use only 16.20.16.144.

```
% setenv RPC_SUPPORTED_NETADDRS 16.20.16.144:16.20.40.139
```

Use two of the available addresses, “16.20.16.144”and “16.20.40.139”.

NOTE: To list more than one network address, use a colon-separated list.

10.5 IDL and ACF Enhancements

This section describes the following enhancements to IDL and the ACF language:

- Automatic binding can use the host's profile
- Enumeration enhancements
- The **client_memory** ACF attribute gives more memory control

10.5.1 Automatic Binding Enhancement

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.

10.5.2 Enumeration in IDL

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 */
```

10.5.3 The client_memory ACF Attribute

While marshaling 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( )**.

10.6 IDL Compiler Enhancements

This section describes the following enhancements to the IDL compiler supported by Gradient DCE for Tru64 UNIX:

- The **-standard** application build options
- Treatment of stub auxiliary files
- Application template feature
- C++ application support

10.6.1 The -standard Build Option

The **-standard_type** IDL compiler command option allows you to specify portable or extended features of the OSF DCE.

The *standard_type* argument specifies which IDL features to enable. If you do not specify this argument, the compiler generates warning messages for all features that are not available in the previous version of OSF DCE.

You can specify one of the following values for the *standard_type* argument:

portable	Allows only the language features available in OSF DCE Version 1.0.2.
dce_v10	**dce_v103** **dec_v10** All are equivalent to the **portable** argument.
dec_v13	Allows all language features supported by the **-standard dce_v10** argument, plus a set of Compaq extensions to its products based on OSF DCE V1.0.3.
dce_v11	Equivalent to **dec_v13**.
extended	Allows all language features supported in the current version of the compiler. This is the default.
dce_v20	Equivalent to the **extended** argument.

The command line in the following example compiles the IDL interface **test.idl** and enables extended features of the OSF DCE:

```
% idl test.idl -standard extended
```

10.6.2 Stub Auxiliary Files

By default, the OSF DCE IDL compiler at V1.0.3 or later does not generate the **-caux** and **-saux** files that V1.0.2 does. However, if you want to use build procedures that work with the V1.0.2 IDL compiler, you can direct the V1.0.3 (or later) IDL compiler to generate empty auxiliary files. To do this, define the environment variable **IDL_GEN_AUX_FILES** as follows:

```
% setenv IDL_GEN_AUX_FILES "1"
```

10.6.3 Generating Application Templates Using the IDL Compiler

The IDL compiler can use your interface definition file to generate a C language template that you can modify to create executable client and server applications. The template feature simplifies the implementation of distributed applications by generating a module of templates for many of the routines that must be provided by the application programmer.

A template is a generated RPC routine that includes the function header and an empty function body. You fill in the function body with application-specific information and integrate the module into your application. By using templates, you can concentrate on the functional aspects of your application program instead of the mechanical process of writing function definitions that match the corresponding IDL definitions.

The template feature is designed to support applications written in the C programming language, and it supports all functions defined in IDL. The template feature does not generate completely executable client and server applications.

To use the IDL compiler to create a template of the manager routines in the server side of your distributed application, specify the **template_manager** option when you compile the application interface module(*filename*.**idl**) with IDL. You can generate a template module containing the client-side routines for your distributed application by specifying the **template_client** option to the IDL compiler.

The **template_manager** compiler option generates a module that contains templates for the routines required in the manager portion of a server application. The **template_client** compiler option generates a module that contains templates for some of the routines used to create the client side of an application. The following IDL compiler command options control generation of template modules.

-template_client *filename*	Directs the IDL compiler to generate a C source file containing a template implementation of each routine that must appear in the client application to use the specified IDL interface. If you do not specify an extension for *filename*, the compiler assigns the file extension **.c**.
-template_manager *filename*	Directs the IDL compiler to generate a C source file containing a template implementation of each routine and operation that must appear in the manager module of the server side of an application to use the specified IDL interface. If you do not specify an extension for *filename*, the compiler assigns the file extension **.c**.

The next table lists every IDL construct that can be defined in a template module. The table indicates whether each construct is specific to a client template, a manager template, or both. If your client application does not use any of the IDL constructs (in the next table) that support client modules, then you will not benefit from using the template feature.

Table 10-1: IDL Constructs Supported by Template Feature

IDL Construct	Template Type
Remote procedure implementations	manager
Context handle rundown routines	manager
[transmit_as] conversion routines	client and manager
[represent_as] conversion routines	client and manager
Customized binding routines	client

Note that remote procedure implementations are defined as all functions defined in IDL for the C programming language.

The template feature is for use during application development, when you might generate template modules repeatedly as you add new functions to an interface. However, after you create and modify the first template module, you should specify a temporary filename for subsequent template modules. Otherwise, you will overwrite the existing modified template module. After creating a new template module in a temporary file, use a text editor to move the new template module into the existing application file which includes the modified templates.

10.6.4 Example of IDL Template Feature

The following sections illustrate how to use the IDL template feature with the **test2** example program included with your DCE software kit. (See *Chapter 13* for more information on programs in this software kit.) The following files are discussed:

- Example interface definition file (**test2.idl**)
- Example manager template (**test2_mgr.c**)

10.6.4.1 Example Interface Definition File

This section shows the **test2.idl** interface definition source code.

```
/*
**COPYRIGHT (C) 1993 BY
**                 DIGITAL EQUIPMENT CORPORATION, MAYNARD
**                  MASSACHUSETTS.  ALL RIGHTS RESERVED.
**

** THIS SOFTWARE IS 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.
** THIS SOFTWARE OR ANY OTHER COPIES THEREOF MAY NOT BE PROVIDED
** OR OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON.  NO TITLE
** TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY TRANSFERRED.
**

** THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE
** WITHOUT NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT
** BY DIGITAL EQUIPMENT CORPORATION.
**
```

```
** DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
** SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
**

**

**  NAME
**

**      test2.idl
**

**  FACILITY:
**

**      RPC Test Program #2
**

**  ABSTRACT:
**

**  Definitions of types/constants and procedures that make up the
**  remote interface to the RPC Test Program #2.
**

*/


[
uuid(eef82780-53bb-11c9-94e0-08002b13d56d),
version(0)
]


interface test2
{


[idempotent] void test2_add
(
    [in]    long        a,
    [in]    long        b,
    [out]   long        *c
);


}
```

10.6.4.2 Example Manager Template

The following IDL command creates a manager template for the server side of the **test2** application interface. The IDL file for the server side of the application is shown in *Section 10.6.4.1 on page 113*.

```
% idl test2.idl -template_manager test2_mgr.c
```

As a result of this command, the compiler generates the **test2_mgr.c** source code template for the **test2.idl** interface definition, shown next. The template begins with comment lines that indicate the version of the IDL compiler that generated the template and the name of the IDL interface for which the template was generated. These comment lines are followed by an include statement for the interface header file and the template routines.

The generated template modules provide the function definition under conditional compilation in both the ANSI-C form and in the older form of the C language. The template module contains both forms of IDL functions so you can use the generated templates on systems on which the C compiler does not yet support the ANSI-C standard.

The following is the **test2_mgr.c** source code template generated by IDL.

```
/* Generated by IDL compiler version DEC DCE T1.0.3-A6 */
/*
** Support routines and Remote Procedure Implementations for interface test2
*/
#include "test2.h"


/*
** Implementation of Remote Procedures for test2
*/


void test2_add
#ifdef IDL_PROTOTYPES
(
    /* [in] */ idl_long_int a,
    /* [in] */ idl_long_int b,
    /*
[out] */ idl_long_int *c
)
#else
(a, b, c)
#endif


#ifndef IDL_PROTOTYPES
    idl_long_int a;
    idl_long_int b;
    idl_long_int *c;
#endif
{
}
```

10.6.4.3 Creating the Executable Manager Program

From the manager template module, you can create the complete, executable server by including the application-specific implementation of the manager routine, as follows:

```
{
*c = a + b;
}
```

10.6.5 C++ Application Support

The **idl** compiler has several options that support the use of C++ language syntax features. The options **-lang** and **-no_cxxmgr** are described in the *Gradient DCE for Tru64 UNIX Reference Guide*.

CHAPTER 11

Application Debugging with the RPC Event Logger

11

11.1 Overview of Debugging Support

The RPC IDL compiler in Gradient DCE for Tru64 UNIX includes enhanced application debugging support beyond the support provided with OSF DCE. The IDL compiler includes the RPC Event Logger — a software utility that records information about operations relating to the execution of an application. Operational information about the program state at a specific point during processing, called an event, is recorded in a file, called an event log. You have the option of directing event logging information to the terminal screen, rather than to a file. In this guide, the terms event log and log are used interchangeably to refer to the stream of logging output captured in the event log file or displayed on the screen.

Event logging provides a detailed, low-level view of the execution of your RPC application. If development of your RPC application is proceeding well, this level of detail may not be necessary. However, when you are in the debugging phase of application development, the continuous execution information provided by the Event Logger and the ability to change the type and timing of logging can be valuable.

11.2 Introduction to the RPC Event Logging Facility

When event logging is enabled, the Event Logger creates one log for each client and server process. To enable the RPC Event Logger, you specify an IDL compiler option that traces events (described in "Enabling Event Logging").

Enabling event logging when compiling allows you the option of generating logs at runtime without rebuilding the application. Once logging is enabled, you can use environment variables and the RPC Log Manager (**rpclm**) to control logging operations. The Log Manager provides a command interface for changing logging operations during application execution.

The RPC Event Logger records events about application calls, context handles, errors, miscellaneous events, and logging operations. These are called event types. Typical RPC events include the following:

- `call_start` — A client application made a call to a server.
- `call_failure` — A client stub terminated abnormally either through an exception or failing status.

- `exception` — An exception was detected in the server stub, and the exception caused the call to terminate.
- `context_rundown` — A context handle on a server was freed by the context rundown procedure.

For application calls, the Event Logger generates events that signal call activation, the call start and end, attempts to rebind to a server, and termination of a server thread.

For context handles, the Event Logger generates events that signal context handle creation and deletion by the client and server, and context handle modification, removal, and rundown.

For errors, the Event Logger generates events that signal call and receive failure from the client, exceptions, server failure, and call transmission failure from the server.

The miscellaneous events provide information about the application manager routine, and input and output argument processing events.

The logging operation itself generates events that display the logging output device, and that signal modification of logging parameters, and event log start and stop.

As a result of using the **-trace** option in the IDL compile command, **idl**, RPC events are generated by code in the client and server stub modules created by the compiler. Note that some events are generated at selected points in the RPC runtime library. For this reason, certain events, such as those relating to the logging operation, are always generated into the application code in addition to the event types you specify.

The events generated in each of these areas are shown in *Table 11-1*. The first column lists events that can be generated, and the second column indicates whether the client or server, or both, can generate the event. See *Section 11.7 on page 133* for a complete description of each event.

Table 11-1: Event Types

Event Name	Origin
Call Events	
activate	server
call_end	client
call_start	client
rebind	client
terminate	server
Context Handle Events	
client_ctx_created	client
client_ctx_deleted	client
client_ctx_destroyed	client
context_created	server

Table 11-1: Event Types (Continued)

context_deleted	server
context_modified	server
context_rundown	server
Error Events	
call_failure	client
exception	server
receive_fault	client
status_fail	server
transmit_fault	server
Miscellaneous Events	
await_reply	client
manager_call	server
manager_return	server
receive	client
Logging Events	
internal_error	client, server
listening	client, server
log_events	client, server
log_file	client, server
log_start	client, server
log_stop	client, server

In the event log, each event is described on a single line divided into five fields. The five fields are defined in the table below.

Table 11-2: Event Log Fields

Field	Field Description
Event Time	The system clock at the time of the event. Events are listed chronologically in the log.
Thread Identity	The hostname, process ID, and thread ID.
Operation Name	The interface and operation name (if available).
Event Name	Name of the event.
Event Data	Data related to the event. This field contains either specific information about logging operations or a string binding that uniquely identifies the client process, server process, or Log Manager process.

The following is an example of an event log generated for an RPC client. The log contains five columns. To improve readability, columns four and five are shown below the first three columns. In addition, the field names have been added to identify the events; the names do not appear in an actual event log. (In subsequent event log examples, the field names are occasionally used instead of actual data to improve readability where necessary.)

```
EVENT TIME                          THREAD IDENTITY   OPERATION NAME


1994-02-07:11:48:18.31.160-5:0010.121  ifdef:8710/1    binopwk.binopwk_add
1994-02-07:11:48:18.32.170-5:0010.121  ifdef:8710/1    binopwk.binopwk_add
1994-02-07:11:48:18.65.180-5:0010.121  ifdef:8710/1    binopwk.binopwk_add


EVENT NAME         EVENT DATA


log_start          all
call_start         ncacn_ip_tcp:16.31.48.109[1821]
call_end
```

This small event log indicates that the following events occurred:

1. The `log_start` event indicates that logging started on February 7, 1994, at 11:48 a.m. on the host named **ifdef**, in process number 8710, and in thread number 1. Event logging was enabled when the **binopwk** interface was compiled with the IDL **-trace** option. The RPC call to the **binopwk_add** operation in the **binopwk** interface caused logging to begin and is the first event logged. The Event Data field indicates that all events are being logged.
2. The `call_start` event indicates an attempt to execute a call to a server. The string binding in the Event Data field shows that the call was made over the TCP/IP transport to host 16.31.48.109 with endpoint 1821. This string binding identifies the server being contacted.
3. The `call_end` event indicates that the RPC call is completed, and control has returned to the caller of **binopwk_add**.

This log indicates that the RPC call to the **binopwk_add** interface was successful because no error events occurred.

11.3 Generating RPC Event Logs

In general, to create an event log you must follow these four basic steps:

1. Specify the **-trace** option in your **idl** command line to enable event logging.
2. Compile and link the application.
3. Assign the event log to a filename or to the screen.
4. Execute the application.

The next sections describe how to use the **-trace** option.

11.3.1 Enabling Event Logging

To enable event logging, specify the **-trace** option when you use the **idl** command to compile an interface. The syntax of the **idl** command with the **-trace** option is as follows:

```
% idl filename -trace value
```

Event types are specified as a value of **-trace**. Valid values and the event types they denote are listed in the table below.

Table 11-3: Event Values and Types

Value	Event Type
all	Log all events
none	Disable all previously specified trace options
calls	Log events relating to all RPC calls
context	Log events relating to context handles
errors	Log errors
misc	Log all miscellaneous events
log_manager	Enable command interface

For more information on the **-trace** option, see *Section 11.3.2 on page 121*.

11.3.2 Using the -trace Option

Once you have used the Event Logger, you will find that it generates a large volume of information to be analyzed. Discard any unneeded log files because the Event Logger will continue to record information in the files, enlarging them until the disk is full.

To help reduce the generation of unwanted information, you can use the **-trace** options to enable event logging on only a subset of events. That is, rather than specifying the **all** option, specify only `calls` or only `context_handles`. The subset you specify will depend on the part of your application you are debugging. Although the **-trace** option provides logging control on a per-compilation basis, the interface must be rebuilt to enable or disable logging of different event types. The **-trace** options offer the ability to select different event types for the various IDL interfaces that might make up a single application.

You can use the **-trace** option to request logging of a single type of event, such as **errors**, with a command similar to the following:

```
% idl binopwk.idl -trace errors
```

You can also use the **-trace** option to request logging of multiple event types, such as **errors** and **calls** as shown below:

```
% idl binopwk.idl -trace errors -trace calls
```

This command enables the Event Logger, specifying error and call event logging.

To enable event logging to trace the execution of RPC calls within a process, perform the following steps:

1 Enable event logging by specifying the **-trace** option in the **idl** command you use to compile each interface definition. This example specifies the **-trace all** option:

```
% idl binopwk.idl -trace all
```

2 Build and link the client and server portions of the application.

3 Use the environment variable RPC_LOG_FILE to direct the log output for both the server and client processes. To store Event Logger output in a file, assign the environment variables to a filename. To direct Event Logger output to the standard terminal output for the process (stdout), assign the environment variable to double quotation marks (""). This guide refers to standard terminal output as the screen.

   In the window from which the server portion of the application will be executed, direct logging for the server to a file with the following syntax:

```
% setenv RPC_LOG_FILE "server.log"
```

   Or, to direct logging for the server to the screen, use the following syntax:

```
% setenv RPC_LOG_FILE ""
```

4 In the window from which the client portion of the application will be executed, direct logging for the client to a file using the following syntax:

```
% setenv RPC_LOG_FILE "client.log"
```

   Or, to direct logging for the client to the screen, use the following syntax:

```
% setenv RPC_LOG_FILE ""
```

Now you can invoke the client and server processes. The event log will be recorded in the specified file or displayed on your screen when you execute the application.

11.3.3 Combining Event Logs

Although event logs are generated locally for each process, you can combine event log files to provide a broader view of application execution.

Note that this section does not give examples of each step in the application development process.

The syntax of a **sort** command is as follows:

```
% sort -m server-filename.log client-filename.log >
client_and_server-filename.log
```

The **-m** option is specified, which indicates that the files are already sorted and prevents reordering of events that occurred at the same time.

If two events have the same timestamp, you receive a warning message after the **sort** is completed.

The following example illustrates how to combine logs from two different systems.

1 The server process command sequence is as follows:

```
% idl fpe_server.idl -trace calls -trace errors
% setenv RPC_LOG_FILE "server.log"
% server
```

2 The client process command sequence is as follows:

```
% idl fpe_server.idl -trace calls -trace errors
% setenv RPC_LOG_FILE "client.log"
% server
```

These command sequences result in two log files: **server.log** and **client.log**, shown below. (Note that, in the following example log files, the Event Data field is not shown.)

This is file **server.log**:

```
1994-03-03:20:37:03.170-5:0010.121 murp:17924/15 fpe.setup   log_start
1994-03-03:20:37:03.170-5:0010.121 murp:17924/15 RPC Log Mgr listening
1994-03-03:20:37:03.180-5:0010.121 murp:17924/15 fpe.setup   activate
1994-03-03:20:37:03.180-5:0010.121 murp:17924/15 fpe.setup   terminate
1994-03-03:20:37:03.200-5:0010.121 murp:17924/15 fpe.float
1994-03-03:20:37:03.200-5:0010.121 murp:17924/15
transmit_fault
1994-03-03:20:37:03.200-5:0010.121 murp:17924/15 fpe.float   terminate
```

This is file **client.log**:

```
1994-03-03:20:37:02.850-5:0010.121 ifdef:28168/1 fpe.stup log_start
1994-03-03:20:37:02.880-5:0010.121 ifdef:28168/1 fpe.stup call_start
1994-03-03:20:37:03.190-5:0010.121 ifdef:28168/1 fpe.stup call_end
1994-03-03:20:37:03.190-5:0010.121 ifdef:28168/1 fpe.flt  call_start
1994-03-03:20:37:03.210-5:0010.121 ifdef:28168/1          receive_fault
1994-03-03:20:37:03.210-5:0010.121 ifdef:28168/1          call_failure
```

3 Next, the two log files are combined and sorted with the **sort** command.

```
% sort -m client.log server.log > client_and_server.log
```

The resulting file **client_and_server.log** is as follows:

```
1994-03-03:20:37:02.850-5:0010.121 ifdef:28168/1 fpe.setup   log_start
1994-03-03:20:37:02.880-5:0010.121 ifdef:28168/1 fpe.setup  call_start
1994-03-03:20:37:03.170-5:0010.121 murp:17924/15 fpe.setup   log_start
1994-03-03:20:37:03.170-5:0010.121 murp:17924/15 RPC Log Mgr listening


1994-03-03:20:37:03.180-5:0010.121 murp:17924/15 fpe.setup   terminate
1994-03-03:20:37:03.190-5:0010.121 ifdef:28168/1 fpe.setup    call_end


1994-03-03:20:37:03.190-5:0010.121 ifdef:28168/1 fpe.float  call_start
1994-03-03:20:37:03.200-5:0010.121 murp:17924/15 fpe.float    activate
1994-03-03:20:37:03.200-5:0010.121 murp:17924/15 fpe.float   exception
```

```
1994-03-03:20:37:03.200-5:0010.121 murp:17924/15        transmit_fault
1994-03-03:20:37:03.200-5:0010.121 murp:17924/15 fpe.float   terminate
1994-03-03:20:37:03.210-5:0010.121 ifdef:28168/1         receive_fault
1994-03-03:20:37:03.210-5:0010.121 ifdef:28168/1          call_failure
```

For the combined output to be accurate, the system clocks on all hosts on which event logs are generated must be closely synchronized. The Distributed Time Service (DTS) component of Gradient DCE for Tru64 UNIX provides such a service. Once the clocks are synchronized, the ordering of events in a combined log file is valid only if the difference between timestamps made on different machines is greater than the inaccuracy field in those timestamps. (See the DTS documentation in the *OSF DCE Administration Guide — Core Components* for more information about timestamps.)

In the preceding **client_and_server.log** file example, consider the event with the timestamp 1994-03-03:20:37:03.180-5:0010.121 and the event that follows it (these two event lines are separated from the rest of the log by a blank line on either side). Note that the timestamps indicate that the `terminate` event precedes the `call_end` event. However, you cannot determine this sequence of events by comparing timestamps because the inaccuracy value at the end of the timestamp is greater than the difference between the timestamps. That is, the difference in time between these events is only 10 milliseconds (the difference between 180 and 190 milliseconds). However, the inaccuracy in the timestamps is 121 milliseconds (10.121). Therefore, the log is not a definitive indicator of which event occurred first. Because of the simplicity of the example and the single thread of control, you can assume that the `terminate` event preceded the `call_end` event.

11.3.4 Disabling Event Logging

To disable event logging, simply compile your interface without specifying the **-trace** option. For example:

```
% idl binopwk.idl
```

11.4 Using Environment Variables and the Log Manager to Control Logging Information

In addition to the **-trace** options, the Event Logger offers two other methods for controlling information in the event log. Each facility is advantageous in different circumstances, depending on the type of processes with which you are working and the type of events you need to log. The two methods are as follows:

- Controlling Logged Events with Environment Variables: Select a subset of event types specified previously with the **-trace** option by creating the environment variable **RPC_EVENTS**. You assign the environment variable to the required event types before executing the process. This method allows you to use event logging without rebuilding the interface; however, you must first stop the process or assign the environment variable before starting it. This method is also useful in cases where you

specified all-inclusive event logging (such as with the **-trace all** option) but you determine while the application is executing that you need only a subset of events.

- Controlling Logged Events with the RPC Log Manager: Select a subset of event types specified previously with the **-trace** option by using the RPC Log Manager command interface. This method allows you to modify event logging parameters for an executing image — there is no need to rebuild the interface or to stop and restart the process. In addition, you can use the Log Manager to modify event types specified with the environment variable **RPC_EVENTS**.

The following sections provide detailed descriptions of how to use each of these methods to control the type of events logged.

11.4.1 Controlling Logged Events with Environment Variables

One way to control the type of events logged is by assigning the environment variable **RPC_EVENTS**. This method is ideal for an application that contains a single RPC interface because environment variables provide control at the process level, rather than at the interface-by-interface level. However, to enable the environment variable you must first stop the client or server process.

To use environment variables to control event logging, first use the IDL **-trace** option in your **idl** compile command and then assign the log file with **RPC_LOG_FILE**. You can then use the environment variable **RPC_EVENTS** to reduce the number of events currently being logged. That is, if you used the **-trace errors** option to request error event logging, you can subsequently use only the environment variable to request logging of **errors** or **none**. You cannot use the environment variable to increase the number of event types to be logged. To do this, you must recompile the interface with the required **-trace** options.

The value of **RPC_EVENTS** is a list of event types separated by commas. The list identifies the event types to be monitored. Valid values are the same as those for **-trace** (except **log_manager**). These values are **all**, **none**, **calls**, **context**, **errors**, and **misc**.

An example command line follows:

```
% setenv RPC_EVENTS "calls,errors"
```

If the environment variable **RPC_EVENTS** was not assigned, then by default all of the events specified with the **-trace** option are written into the event log.

11.4.2 Controlling Logged Events with the RPC Log Manager

During application development, certain problems occur only after a server has executed some number of calls. Other problems may require more information than anticipated to debug. These problems can be addressed by enabling the RPC Log Manager in your application image. The Log Manager offers a command line interface (**rpclm**) for manipulating logging operations

during application execution. When you use the **rpclm** command line interface, you need not rebuild your interface or stop and restart your server or client process to manipulate logging operations.

The **rpclm** commands are shown in *Table 11-4*.

Table 11-4: Command Interface to rpclm

Command	Description
inquire	Inquire about the currently logged events and determine the name of the active log file.
log	Specify additional events to log. Valid values are **all**, **none**, **calls**, **context**, **errors**, and **misc**.
unlog	Disable logging of the specified event types. Valid values are **all**, **none**, **calls**, **context**, **errors**, and **misc**.
file	Change the output device or file to which events are logged.
quit	Terminate the **rpclm** session.
help	Display a description of **rpclm** commands.

Follow these steps to enable the RPC Log Manager to control event logging:

1 Use the **-trace log_manager** option in your **idl** compile command.

2 Create the **RPC_LOG_FILE** environment variable and assign it to a filename or to screen output.

3 Execute the client or server process, or both.

4 When the first call is made to an interface compiled with the **-trace** option, a `listening` event will be generated into the event log. Invoke the **rpclm** command interface (as specified in step 4 below) by specifying the string binding from the `listening` event.

NOTE: Only string bindings from a `listening` event can be used to invoke **rpclm**.

The **rpclm** command interface allows you to control event logging parameters from your keyboard. You can use the command interface to reduce the events currently being logged as well as to manipulate logging operations. You can enable or disable logging of different event types (within the set selected with the **-trace** option), store event logging in a file or display it on the screen, inquire about the current event types being logged, and display the name of the current log file.

The following procedure illustrates how to use the Log Manager:

1 When you compile your interface with the **idl** compile option, include the **-trace log_manager** option. For example:

```
% idl binopwk.idl -trace all -trace log_manager
```

2 Assign the **RPC_LOG_FILE** environment variable to a filename. For example:

```
% setenv RPC_LOG_FILE "client.log"
```

3 Execute the client or server process, or both.

4 Upon the first remote procedure call to an interface compiled with the **-trace log_manager** option, a `listening` event will be generated into the log. Examine the Event Data field of the `listening` event in the log to determine the Log Manager string binding. (The RPC Event Logger is itself a client/server application: the Log Manager is a server process, and **rpclm** is its client. The **rpclm** client uses the string binding of the `listening` event to communicate with the Log Manager server.) Invoke **rpclm** and specify the Log Manager string binding. For example, consider the following event:

```
murp:17868/15  RPC Log Mgr  listening ncacn_ip_tcp:16.31.48.144[3820]
```

The `listening` event indicates that the RPC Log Manager is waiting for commands from **rpclm**. (Note that, in the example, the Time field is not shown.) To invoke **rpclm**, enter the `listening` event string binding for this server process from the Event Data field as follows:

```
% rpclm "ncacn_ip_tcp:16.31.48.144[3820]"
```

NOTE: You must enclose the string binding in double quotation marks ("").

---

5 As you execute **rpclm** commands, the Log Manager displays current logging parameters that indicate the changes made to event logging for this process. For example:

```
rpclm> >unlog all
Event types:
Events logged to terminal
rpclm> log calls
Event types: calls
Events logged to terminal
```

The log for this server process will have corresponding events logged as follows:

```
<time>  murp:17868/15 RPC Log Mgr   log_events      none
<time>  murp:17868/15 RPC Log Mgr   log_events      calls
```

The following example illustrates a command dialog between the user and **rpclm**. The dialog begins when the user specifies a string binding from a `listening` event to **rpclm**.

```
% rpclm "ncacn_ip_tcp:cltdce[1821]"
rpclm> help
rpclm Commands:
inquire - Display logged events and log filename
log     - Specify additional events to log
unlog   - Specify events that should no longer be logged
file    - Change file into which events are logged
quit    - Exit log manager
```

```
rpclm> inquire
Event Types: calls
Events logged to terminal
rpclm> log errors
Event Types: calls errors
Events logged to terminal
rpclm> file server.log
Event Types: calls errors
Events logged to file 'server.log'
rpclm> quit
```

In this dialog, the user enters the **help** command to display the **rpclm** commands and command descriptions.

The user then enters the **inquire** command to display the types of events being logged and the log filename. In this example, errors are being logged to the screen.

The user enters the **log calls** command to specify that the Log Manager should start logging all events relating to calls, in addition to error events.

The user then enters the **file** command and specifies a filename. This command requests that **rpclm** change its output device from the terminal screen to a file named **server.log**.

The user then enters the **quit** command to terminate the **rpclm** session.

11.5 Using the -trace Option, Environment Variables, and the Log Manager Together

This section describes a few different ways to use the **-trace** options, environment variables, and the Log Manager together. When you are learning to use the Event Logger, one possible approach is to specify all-inclusive event logging with the **-trace all** IDL compilation option, and then examine the event log to get an understanding of typical output. You can then use the environment variable **RPC_EVENTS** to log only those events needed, such as **calls** or **errors**.

In the case of a running process that you do not want to terminate, use a different method. First enable the Event Logger, specifying logging of all events, and enable the Log Manager also, as follows:

```
% idl filename -trace all -trace log_manager
```

Set the event log to display on the screen, as follows:

```
% setenv RPC_LOG_FILE ""
```

Then, assign the RPC_EVENTS environment variable so it will not log any event types, as follows:

```
% setenv RPC_EVENTS "none"
```

With these parameters set, the only event that will be displayed is the `listening` event once the first call is made to a server interface compiled with the **-trace log_manager** option. You can then obtain the string binding for the process and use it later, if needed. Once you start the process, if an error occurs, use the string binding to invoke the **rpclm** command interface and log the needed events. Any **rpclm** commands issued at this point will modify the

RPC_EVENTS environment variable assignment. For example, if you assign the environment variable **RPC_EVENTS** to **calls** and then issue a command to **rpclm** to **log errors**, errors as well as calls will be logged.

Once you are familiar with Event Logger output, consider regularly using the command interface to enable or disable subsets of event types as needed.

This section provides an example of common tasks you may need to perform during event logging. In this particular example, a distributed server process provides a mathematical calculation service. The client process passes data to be calculated to the server process. This type of processing often generates exception events such as those in the example event log. That is, some operations are interrupted by `floating point overflow` and `integer division by zero` exceptions, as well as others. This example uses **rpclm** to control logging of a server process; however, **rpclm** can also be used to control event logging for a client process.

The following processes are shown in three windows: a server process window, a client process window, and an **rpclm** window.

1 **Server Window**: The user first enables the RPC Event Logger by specifying the **-trace all** and **-trace log_manager** options in the **idl** command line:

% **idl server.calc -trace all -trace log_manager**

2 **Server Window**: The user starts the server process. The server receives a client call and initializes the RPC Log Manager. The environment variables were assigned to enable event logging with no event types selected, so only Log Manager events are output, as shown. Note that the endpoint displayed for the `listening` event is the endpoint of the Log Manager. (The time field is not shown.)

```
% setenv RPC_LOG_FILE  ""
% setenv RPC_EVENTS "none"
% server ncacn_ip_tcp

murp:17868/15 fpe.setup     log_start  none
murp:17868/15 RPC Log Mgr listening ncacn_ip_tcp:16.31.48.144[3820]
```

3 **Client Window**: The user invokes the client process. The specified string binding is used to find the server. The client process displays the output `PASS 1` upon completion.

```
% Client ncacn_ip_tcp 16.31.48.86 [3123]
PASS 1
```

4 **rpclm Window**: The user invokes **rpclm** and specifies the string binding displayed in the `listening` event output by the server process, shown in step 2. The string binding must be enclosed in double quotation marks (""). The user issues the **inquire** command, and the event logging parameters for the server process are displayed. The Log Manager reply indicates that no event types are enabled and that the event log is being displayed on the screen from which the server process was started. The user issues the **log errors** command to enable logging of error events for the server process.

```
% rpclm "ncacn_ip_tcp:16.31.48.144[3820]"
```

```
rpclm> inquire
Event types:
Events logged to terminal
rpclm> log errors
Event types: errors
Events logged to terminal
```

5 **Client Window**: The user invokes the client process a second time. The error events that occur during server execution are logged to the server window. The client process displays the output `PASS 2` upon completion.

```
% Client ncacn_ip_tcp 16.31.48.86 [3123]
PASS 2
```

6 **Server Window**: The server process receives the command from **rpclm** to start logging errors. Any errors that occur in the server process are logged. (The time field is not shown.)

```
murp:17868/15 RPC Log Mgr       log_events     errors
murp:17868/15 fpe.flt_overflw   exception      Floating point
                                                 overflow (dce/thd)
murp:17868/15                   transmit_fault rpc_s_fault_fp_overflow
murp:17868/15 fpe.flt_underflw  exception      Floating point
                                                 underflow (dce/thd)
murp:17868/15                   transmit_fault rpc_s_fault_fp_underflow
murp:17868/15 fpe.flt_divbyzer  exception      Floating point/decimal
                                               divide by zero (dce/thd)
murp:17868/15                   transmit_fault rpc_s_fault_fp_div_by_zero
murp:17868/15 fpe.dble_overflw  exception      Floating point
                                                 overflow (dce/thd)
murp:17868/15                   transmit_fault rpc_s_fault_fp_overflow
murp:17868/15 fpe.dble_underflw exception      Floating point
                                                 underflow (dce/thd)
murp:17868/15                   transmit_fault rpc_s_fault_fp_underflow
murp:17868/15 fpe.dble_divbyzer exception      Floating point/decimal
                                                 divide by zero (dce/thd)
murp:17868/15                   transmit_fault rpc_s_fault_fp_div_by_zero
```

7 **rpclm Window**: The user issues the **unlog all** command to disable logging of all previously specified event types.

```
rpclm> unlog all
Event types:
Events logged to terminal
```

8 **Server Window**: The event log now contains an entry that indicates the Event Logger will stop logging previously specified events.

```
<time>  murp:17868/15   RPC Log Mgr    log_events       none
```

9 **rpclm Window**: The user issues a **log calls** command to enable logging of call events.

```
rpclm> log calls
Event types: calls
Events logged to terminal
```

10 **Server Window**: The newest event log entry indicates that the Event Logger will start logging call events.

```
<time>   murp:17868/15   RPC Log Mgr     log_events     calls
```

11 **rpclm Window**: Because logging output will increase now that call events are being logged, the user issues an **rpclm** command to redirect logging output to a file named **server_calc.log**. When the application terminates and logging is complete, the user can use a text editor to view and search for entries in the log. This log file will contain only those call events from the server process.

```
rpclm> file server_calc.log
Event types: calls
Events logged to file 'server_calc.log'
```

12 **Server Window**: The newest event log entry indicates that the logger will start redirecting logging information to file **server_calc.log**.

```
<time>  murp:17868/15  RPC Log Mgr    log_file      server_calc.log
```

13 **ClientWindow**: The user invokes the client process a third time. The call events that occur during server execution are logged to file **server_calc.log**. The client process displays the output `PASS 3` upon completion.

```
% Clientncacn_ip_tcp 16.31.48.86 [3123]
PASS 3
```

14 **Server Log**: This is log file **server_calc.log** (the time field is not shown):

```
% more server_calc.log

murp:17868/15 RPC Log Mgr       log_start server_calc.log
murp:17868/15 fpe.setup         activate ncacn_ip_tcp:16.31.48.109[2905]
murp:17868/15 fpe.setup         terminate ncacn_ip_tcp:16.31.48.109[2905]
murp:17868/15 fpe.flt_overflw   activate  ncacn_ip_tcp:16.31.48.109[2905]
murp:17868/15 fpe.flt_overflw   terminate
murp:17868/15 fpe.flt_underflw  activate  ncacn_ip_tcp:16.31.48.109[2905]
murp:17868/15 fpe.flt_underflw  terminate
murp:17868/15 fpe.flt_divbyzer  activate  ncacn_ip_tcp:16.31.48.109[2905]
murp:17868/15 fpe.flt_divbyzer  terminate
murp:17868/15 fpe.dble_overflw  activate  ncacn_ip_tcp:16.31.48.109[2905]
murp:17868/15 fpe.dble_overflw  terminate
murp:17868/15 fpe.dble_underflw activate  ncacn_ip_tcp:16.31.48.109[2905]
murp:17868/15 fpe.dble_underflw terminate
murp:17868/15 fpe.dble_divbyzer activate  ncacn_ip_tcp:16.31.48.109[2905]
murp:17868/15 fpe.dble_divbyzer terminate
```

15 **rpclm Window**: The user issues a **file** command to redirect event logging output from **server_calc.log** to the terminal screen. To do this, press the Return key without specifying a filename when the Log Manager prompts for one.

```
rpclm> file
New File Name: <Return>
Event types: calls
Events logged to terminal
rpclm>
```

16 **Server Window**: The final event in the **server_calc.log** file is a `log_file` event, which indicates that event logging output is being redirected, in this case to the terminal screen. Therefore, no filename is displayed to the right of the event name.

```
<time>    murp:17868/15    RPC Log Mgr    log_file
```

11.6 Using Event Logs to Debug Your Application

The RPC Event Logger is designed to help you debug your distributed application and is an enhancement over the basic diagnostics in the RPC product. The diagnostics alone provide minimal information. For example, the sample program called **test2**, which is provided with the Gradient DCE for Tru64 UNIX, generates the `rpc_x_no_more_bindings` exception when the client fails to contact the server. Without the aid of RPC event logging, this is the only diagnostic information available.

The following example shows the basic RPC diagnostic information that an application displays when an error occurs.

```
% test2

*** Unable to obtain server binding information
Make sure environment variable RPC_DEFAULT_ENTRY = .:/test2_server
Exception: no more bindings (dce / rpc)
IOT trap (core dumped)
```

If you enable RPC event logging by defining the environment variable RPC_LOG_FILE, then the details of client execution can be captured in a file. From the event log, you can determine which servers the client tried to contact and the reason each attempt failed.

In the following event log example, the Event Data field on the `rebind` events indicates that the interface is not registered in the endpoint map and that a communications failure occurred. This information indicates that the server either is not running or it failed to register properly with the endpoint mapper.

The final event, `call_failure`, indicates that the call was terminated with the `no more bindings` status. This event indicates that the client tried all available servers but failed to communicate with any of them. (Note that in the first column the word *time* represents the actual value for time.)

```
% test2

time ko:11436/1 test2.test2_add  log_start   all
time ko:11436/1 test2.test2_add  call_start  ncacn_ip_tcp:16.20.16.27[]
time ko:11436/1 test2.test2_add  rebind      not registered in endpoint
                                              map(dce/rpc)
time ko:11436/1 test2.test2_add  call_start  ncacn_dnet_nsp:4.262[]
time ko:11436/1 test2.test2_add  rebind      not registered in endpoint
                                              map(dce/rpc)
time ko:11436/1 test2.test2_add  call_start  ncadg_ip_udp:16.20.16.27[]
time ko:11436/1 test2.test2_add  rebind      comm failure (dce/rpc)
time ko:11436/1 call_failure                 no more bindings (dce/rpc)
*** Unable to obtain server binding information
Make sure environment variable RPC_DEFAULT_ENTRY = .:/test2_server
Exception: no more bindings (dce / rpc)
IOT trap (core dumped)
```

11.7 Event Names and Descriptions

This section lists and describes RPC events. See the table in *Section 11.2 on page 117* for a list of events by type (calls, context handles, errors, miscellaneous, and logging) and their origin (client or server).

activate	A thread was assigned to process an RPC call on a server, and the server stub has started processing input arguments. The Event Data field of the event log contains the string binding of the client application making the call.
await_reply	The transmission of input arguments in a call from a client application to a server is completed. The event is generated by the client stub. The client application is waiting for output arguments from the server.
call_end	A call from a client application is complete and the client stub is returning to the caller.
call_failure	A client stub terminated abnormally because either an exception occurred or a failing status was returned. The Event Data field of the event log contains the error text associated with the exception or RPC status code.
call_start	A client application attempted to make a call to a server. The event is generated by the stub within the client application. The Event Data field of the event log displays the string binding of the server being contacted.
client_ctx_created	A client application has allocated a context handle on a particular server. The Event Data field of the event log contains the following information about this event: ■ The address representing the context handle in the client address space (an opaque pointer) ■ The UUID which can be used to identify the corresponding context handle on the server ■ The string binding of the server on which the actual context resided
client_ctx_deleted	The client application representation of a context handle is being deleted to reflect the deletion of the context handle on the server. The Event Data field of the event log contains the following information about this event: ■ The address representing the context handle in the client address space (an opaque pointer) ■ The UUID which can be used to identify the corresponding context handle on the server ■ The string binding of the server on which the actual context resided
client_ctx_destroyed	A client application has destroyed the client representation of a context handle through the **rpc_ss_destroy_client_context( )** routine. The Event Data field of the event log contains the following information about this event: ■ The address representing the context handle in the client address space (an opaque pointer) ■ The UUID which can be used to identify the corresponding context handle on the server ■ The string binding of the server on which the actual context resided

context_created	A new context handle was created on a server and returned from the application manager routine. The Event Data field of the event log contains both the application value of the context handle and the UUID assigned to represent this context handle.
context_deleted	A context handle on a server has been deleted by the application manager routine. The Event Data field of the event log contains both the application value of the context handle and the UUID assigned to represent this context handle.
context_modified	A context handle on a server was returned from the application manager routine with a value that is different from its previous value. The Event Data field of the event log contains both the application value of the context handle and the UUID assigned to represent this context handle.
context_rundown	A context handle on a server was freed by the context rundown procedure. The Event Data field of the event log contains both the application value of the context handle and the UUID assigned to represent this context handle.
exception	An exception was detected in the server stub, and the exception caused the call to terminate. The Event Data field of the event log contains a text description of the exception.
internal_error	A failure occurred in the support routines that manage the Event Logger. Check the Event Data field of the event log for a description of the cause of the event. If the error does not seem to indicate a transient network problem or an environmental failure, report the failure in a Software Performance Report (SPR).
listening	The RPC Log Manager has started to listen for **rpclm** commands. The `listening` event is generated by the portion of the RPC Log Manager built into your application by the RPC runtime when you specify the **-trace log_manager** option on your IDL compilation. The RPC Log Manager services the requests generated by the **rpclm** command. You use one of the string bindings from a `listening` event to invoke the **rpclm** command interface.
log_events	Event logging was modified through the Log Manager command interface **rpclm**. The Event Data field of the event log contains the new set of events being logged.
log_file	Event logging was modified through the Log Manager command interface **rpclm**. The Event Data field of the event log contains the new filename for the event log. If no filename is displayed, events are being logged to the screen.
log_start	A new event log was created or event logging was resumed after being suspended by a user command to the Log Manager command interface **rpclm**. The Event Data field in the event log contains a list of event types being logged.
log_stop	Event logging was stopped through the Log Manager command interface **rpclm**.
manager_call	The server stub is about to call the application manager routine.
manager_return	Control has just returned from the application manager routine to the server stub.
rebind	A call from a client application to a server failed. The Event Data field in the event log shows the reason for the failure to contact the server. The event is generated by the stub within the client application. The call failed on an auto_handle operation and the client is attempting to rebind to the next server.

receive	Following the transmission of input arguments from a client application call to a server, the client received a reply and has started processing output arguments.
receive_fault	The client received a fault indicating a failure on the server. The Event Data field of the event log contains the RPC status that identifies the failure. All failures have fault codes which you can find in the file **ncastat.idl**. If the fault code in the **ncastat.idl** file is too general (such as `unspecified fault`), examine the server event log for precise failure information.
status_fail	A failure status was encountered in the server stub. The Event Data field of the event log describes the failure.
terminate	The server thread has completed processing the call and has terminated.
transmit_fault	The server runtime is sending fault information to the client application. The Event Data field of the event log indicates the name of the fault being sent. The fault information in this field is listed in the **ncastat.idl** file. The fault information in this field may be less descriptive than the information logged about the actual error. (See the `exception or status_fail` events in the event log to obtain precise failure information.)

11.8 Summary

The RPC Event Logger is a developer's aid for debugging DCE RPC applications. The RPC Event Logger allows you to modify IDL-generated stub routines in order to generate event logs of runtime execution of RPC calls on the screen or in a file. In addition, the RPC Log Manager command interface **(rpclm)** provides command line access to event logging parameters, allowing you to enable and disable debugging support of clients and servers as they execute.

The DCE RPC application development environment is designed to create applications that are portable to other DCE platforms and that can interoperate with other DCE applications. Use of Gradient DCE for Tru64 UNIX RPC Event Logger does not affect code portability or interoperability. Because the Event Logger does not modify the application, you can take advantage of event logging without affecting application portability to other hardware or software platforms.

In addition, use of Gradient DCE for Tru64 UNIX RPC Event Logger does not limit interoperability with other DCE implementations. Because event logs are generated only in the local application, communication protocols are not modified. You can, for example, use the event logging facility with any server process running under Gradient DCE for Tru64 UNIX or with any client process communicating with an RPC server on any hardware or software platform.

CHAPTER 12

Developing Distributed Applications with FORTRAN

12

12.1 Overview of Applications with FORTRAN

This chapter explains how to use DIGITAL FORTRAN® in the development of distributed applications that make remote procedure calls.

This chapter provides the following information:

- Interoperability and portability issues as they relate to applications written in DIGITAL FORTRAN
- A comprehensive example that introduces and illustrates several concepts
- General reference information about DIGITAL FORTRAN and remote procedure calls, including a discussion about restrictions

12.2 Interoperability and Portability

In general, an application you create in the Gradient DCE for Tru64 UNIX RPC environment will interoperate with other DCE RPC applications and will port to other DCE platforms if it complies with the appropriate programming language standards. More specifically:

- Any client that you have correctly created in a Gradient DCE for Tru64 UNIX RPC environment to use a DCE interface expressed in an IDL file will interoperate with any DCE RPC server that supports the interface.
- Any server that you have correctly created in a Gradient DCE for Tru64 UNIX RPC environment to use a DCE interface expressed in an IDL file will interoperate with any DCE RPC client that makes calls on the interface.

Typically, applications created in the DCE RPC environment are written in the C programming language. However, if you use the DIGITAL FORTRAN support in Gradient DCE for Tru64 UNIX, the application will be subject to the following portability constraint:

- Gradient DCE for Tru64 UNIX RPC applications that contain code written in DIGITAL FORTRAN in a Tru64 UNIX environment and that use a DCE interface expressed in an IDL file will interoperate with any corresponding DCE server or DCE client. However, you can port these applications only to other Gradient DCE for Tru64 UNIX environments.

12.3 Remote Procedure Calls Using FORTRAN — Example

The Gradient DCE for Tru64 UNIX IDL compiler provides similar support for applications written in DIGITAL FORTRAN as that provided for applications written in C. That is, you can write an RPC client in DIGITAL FORTRAN or you can write one or more manager routines in the server side of the application in DIGITAL FORTRAN. If you are unfamiliar with the tasks involved in developing an RPC application, see the chapter about application building in the *OSF DCE Application Development Guide*.

The DIGITAL FORTRAN support consists of stubs that use DIGITAL FORTRAN linkage conventions and a file that contains DIGITAL FORTRAN definitions of the constants and types declared in an interface definition. (These conventions and definitions are explained in Remote Procedure Calls Using FORTRAN — Example.)

The following sections present a comprehensive example that demonstrates how you can create the various parts of a simple, distributed payroll application using DIGITAL FORTRAN.

The important features of this example are as follows:

- The example client application reads time-card information, passes it to a server that calculates wages, and prints the results.
- Both the client and the portion of the server that calculates gross pay (the manager routine) are written in DIGITAL FORTRAN.
- The initialization portion of the server application is written in C.

12.3.1 Where to Obtain the Example Application Files

All of the example application files referenced in this chapter are located in the following directory in your kit:

/usr/examples/dce/rpc/payroll

The next table lists application files that normally would be created by the programmer for an application. To demonstrate application building, these application files are provided for you in the software kit. The second table in Compiling the Interface with the IDL Compiler lists the files generated by the IDL compiler for the example application.

Before you execute any of the example compilations, builds, or run commands in this chapter, copy all of the files listed in the first table to an empty directory. Entegrity recommends that you read the file named README in the same subdirectory. Then build and run the examples.

Table 12-1: Example Files Created by the Programmer

Filename	File Description
payroll.idl	The interface definition file that contains the application programming interface (API) to the remote procedure call **calculate_pay( )**.
print_pay.for	The FORTRAN source file for the client side of the application.
server.c	The FORTRAN source file that contains the initialization code for the server side of the application.
manager.for	The FORTRAN source file for the server side of the application.
Makefile.unix	The description file that builds the example application.
payroll.dat	The data input file for the example application.

The programs, procedures, and data files in the payroll example should be the same in this chapter and in the specified subdirectory that came with your Gradient DCE for Tru64 UNIX software kit. For example, file **payroll.idl** as it appears in The Interface File and Data File (payroll.idl and payroll.dat) should be identical to the following file:

/usr/examples/dce/rpc/payroll/payroll.idl

For all of the example files, if there is a difference between the file as shown in this chapter and the file in the subdirectory, assume that the file in the subdirectory is the correct one.

12.3.2 The Interface File and Data File (payroll.idl and payroll.dat)

The following interface, named **payroll.idl**, is part of the example application. The name of the remote procedure in the interface is **calculate_pay( )**. The interface does not indicate that this procedure is written in DIGITAL FORTRAN.

```
/*
**  Copyright (c) 1993 by
**      Digital Equipment Corporation, Maynard, Mass.
**
*/


[
uuid(d1b14181-6543-11ca-ba11-08002b17908e),
version(1.0)
]
interface payroll
{
    const long string_data_len = 7;
```

```
    typedef struct {
        [string] char grade[string_data_len + 1];
          /*Storage for string must include space for null terminator*/
        short   regular_hours;
        short   overtime_hours;
    } timecard;


    void calculate_pay(
        [in] timecard cards[1..7],
        [out] long *pay
    );
}
```

The next part of the example is the data file **payroll.dat**, which the client side of the application reads. The facts about each employee appear in 8 records. The first record contains the employee's name (40 characters) and grade (7 characters). Records 2 to 8 contain the number of regular hours and overtime hours worked on Monday to Sunday.

NOTE: The timecard structure defined in **payroll.idl** does not specify the employee's name in the data going to the remote procedure.

```
Jerry Harrison                          FOREMAN
 8 1
 8 1
 8 2
 8 2
 8 1
 0 4
 0 0
Tony Hardiman                           WORKER
 8 0
 8 0
 8 0
 8 2
 8 0
 0 4
 0 0
Mary Flynn                              WORKER
 8 1
 8 1
 8 2
 8 0
 8 1
 0 4
 0 0
```

12.3.3 Compiling the Interface with the IDL Compiler

To compile an RPC interface, you must use the **idl** command to invoke the IDL compiler. To compile an RPC interface for a DIGITAL FORTRAN application, you must select the following IDL options:

- Option **-lang fortran**. This option specifies FORTRAN as the source code language.

- Option -**standard extended**. This option enables features beyond those available in OSF DCE Version 1.0.3.

The following example command illustrates how to invoke the IDL compiler to compile the sample DIGITAL FORTRAN application interface:

```
% idl payroll.idl -lang fortran -standard extended
```

As a result of this command, the IDL compiler generates the files listed in the next table.

Table 12-2: Example Files Created by IDL

Filename	File Description
payroll_cstub.o	The stub file generated by the IDL compiler for the client side of the application.
payroll_sstub.o	The stub file generated by the IDL compiler for the server side of the application.
payroll.for	An include file that emulates the C language header file (**.h**) and that documents the valid syntax for subroutine calls that are used in the FORTRAN source files. This file will be called out in **Makefile.unix** and linked with the other application files because it refers to constants and types defined in the interface definition.
payroll.for_h	A file generated by the IDL compiler that is used to build the stub files.

File **payroll.for**, as generated by the IDL compiler, is next.

```
C     Generated by IDL compiler version DEC DCE Vn.n.n-n
C
C     The following statements must appear in application code
C     INCLUDE 'NBASE.FOR'


      INTEGER*4 STRING_DATA_LEN
      PARAMETER (STRING_DATA_LEN=7)


      STRUCTURE /TIMECARD/
          CHARACTER*8 GRADE
          INTEGER*2 REGULAR_HOURS
          INTEGER*2 OVERTIME_HOURS
      END STRUCTURE


C      SUBROUTINE CALCULATE_PAY(CARDS, PAY)
C      RECORD /TIMECARD/ CARDS(7)
C      INTEGER*4 PAY
```

As you read this chapter, it is important to remember that the interface defined in file **payroll.idl** appears as DIGITAL FORTRAN statements in file **payroll.for**. As a specific instance, consider the overtime hours field. Its

definition appears in **payroll.idl** as the statement `short overtime_hours`, and in **payroll.for** as the statement `INTEGER*2 OVERTIME_HOURS`. The overtime hours data in file **payroll.dat** is read into a data item of this type.

12.3.4 The Client Application Code for the Interface (print_pay.for)

Suppose that the directory in which the interface was compiled also contains file **print_pay.for**. This is the source file for the client side of the distributed application. Its contents follow.

```
CThis is the client side of a payroll application that
C   uses remote procedure calls.
C
        PROGRAM PRINT_PAY
        INCLUDE 'PAYROLL.FOR'    ! Created by the IDL compiler from
                                 !    file PAYROLL.IDL.
CCOPYRIGHT (C) 1993 BY DIGITAL EQUIPMENT CORP., MAYNARD MASS.
C       The structure of a time card is described in the included file.
        RECORD /TIMECARD/ CARDS(7)
        CHARACTER*40 NAME
        CHARACTER*8  GRADE
        INTEGER*4 PAY
        INTEGER*4 I
C
CRead eight records for the current employee.
   10   READ (4, 9000, END=100) NAME, GRADE  ! First record
 9000   FORMAT (A40, A8)
        DO 20 I = 1, 7   ! Second through eighth records
           READ (4,9010) CARDS(I).REGULAR_HOURS, CARDS(I).OVERTIME_HOURS
 9010      FORMAT (I2, I2)
           CARDS(I).GRADE = GRADE
   20   CONTINUE
C
CCall remote procedure CALCULATE_PAY to calculate the gross pay.
        CALL CALCULATE_PAY (CARDS, PAY)
CDisplay the current employee's name and gross pay.
        WRITE  (6, 9020) NAME,     PAY
 9020   FORMAT (1X,      A40,  1X, I4  )
        GO TO 10
C
  100   STOP
C
        END
```

To compile and link the client program **print_pay.for**, which at runtime makes remote procedure calls to a server that supports the payroll interface, use the following commands.

```
% fortran -c print_pay.for

% ld -o print_pay print_pay.o payroll_cstub.o

-lfor -lutil -lUfor -lm -lots -ldce -lpthreads -lmach -lc_r -lm
```

Instead of using these two commands directly to build the client part of the application, you can use **make** to build the entire application using the supplied Makefile, called **Makefile.unix**. See *Section 12.3.8 on page 146* and *Section 12.3.9 on page 148* for information about building and running this example.

This program reads its data from DIGITAL FORTRAN logical unit 4. A command in **Makefile.unix** defines the logical unit.

12.3.5 The Server Initialization File (server.c)

Because all programming interfaces to the RPC runtime are specified in C, you must write the code that sets up the server in C. In this example, the server setup code (also called the initialization code) is in file **server.c, s**hown next.

```
/* This is program SERVER.C that sets up the server for
   the application code whose origin is FORTRAN
   subroutine CALCULATE PAY.               */


/*
** Copyright (c) 1993 by
**     Digital Equipment Corporation, Maynard, Mass.
**
*/


#include <stdio.h>
#include <file.h>
#include <dce/dce_error.h>
#include "payroll.for_h"  /* The IDL compiler created this file from
                             file PAYROLL.IDL.                    */


static char error_buf[dce_c_error_string_len+1];
static char *error_text(st)
    error_status_t st;
{
    error_status_t rst;
    dce_error_inq_text(st, error_buf, &rst);
    return error_buf;
}


main()
{
  error_status_t st;
  rpc_binding_vector_p_t bvec;


  /* Register all supported protocol sequences with the runtime. */
  rpc_server_use_all_protseqs(
        rpc_c_protseq_max_calls_default,
        &st
  );
  if (st != error_status_ok)
  {
   fprintf(stderr, "Can't use protocol sequence - %s\n",error_text(st));
```

```
 exit(1);
}


  /* Register the server interface with the runtime.               */
  rpc_server_register_if(
      payroll_v1_0_s_ifspec,  /* From the IDL compiler;             */
                              /* "v1_0" comes from the statement    */
                              /* "version(1.0)" in file PAYROLL.IDL.*/
      NULL,
      NULL,
      &st
  );
  if (st != error_status_ok)
  {
      printf("Can't register interface - %s\n", error_text(st));
      exit(1);
  }


  /* Get the address of a vector of server binding handles.  The
     call to routine rpc_server_use_all_protseqs() directed the
     runtime to create the binding handles.                      */
  rpc_server_inq_bindings(&bvec, &st);
  if (st != error_status_ok)
  {
      printf("Can't inquire bindings - %s\n", error_text(st));
      exit(1);
  }


  /*Place server address information into the local endpoint map.*/
  rpc_ep_register(
      payroll_v1_0_s_ifspec,
      bvec,
      NULL,
      (idl_char*)"FORTRAN Payroll Test Server",
      &st
  );
  if (st != error_status_ok)
  {
      printf("Can't register ep - %s\n", error_text(st));
  }


  /* Place server address information into the name service database. */
  rpc_ns_binding_export(
      rpc_c_ns_syntax_default,
      (idl_char*)".:/FORTRAN_payroll_mynode",
      payroll_v1_0_s_ifspec,
      bvec,
      NULL,
      &st
  );
  if (st != error_status_ok)
  {
```

```
        printf("Can't export to name service - %s\n", error_text(st));
    }


    /* Tell the runtime to listen for remote procedure calls.
       Also, FORTRAN cannot support multiple threads of execution. */
    rpc_server_listen((int)1, &st);
    if (st != error_status_ok)
        fprintf(stderr, "Error listening: %s\n", error_text(st));


}
```

12.3.6 The Server Application Code for the Interface (manager.for)

The server application code, written in DIGITAL FORTRAN, is declared in file **payroll.idl** as **calculate_pay( )**. File **manager.for** contains subroutine **calculate_pay** as follows:

```
        SUBROUTINE CALCULATE_PAY(CARDS, PAY)
        INCLUDE 'PAYROLL.FOR'    ! Created by the IDL compiler from
                                 ! file PAYROLL.IDL.
C
CCOPYRIGHT (C) 1993 BY DIGITAL EQUIPMENT CORP., MAYNARD MASS.
C       The structure of a time card is described in included
C       file PAYROLL.FOR.


        RECORD /TIMECARD/ CARDS(7)
        INTEGER*4 PAY
        INTEGER*4 I


        PAY = 0
        DO 10 I = 1, 7
C    The basic hourly wage is $6.00.
            PAY = PAY + 6 * CARDS(I).REGULAR_HOURS
C    The following comparison does not include last character
C       of GRADE, because it arrives as a null terminator.
            IF (CARDS(I).GRADE(1:STRING_DATA_LEN) .EQ. 'FOREMAN') THEN
C        The overtime hourly wage for a foreman is $12.00.
                PAY = PAY + 12 * CARDS(I).OVERTIME_HOURS
            ELSE
C        The overtime hourly wage for a worker is $9.00.
                PAY = PAY +  9 * CARDS(I).OVERTIME_HOURS
            END IF
   10   CONTINUE


        RETURN
        END
```

To create the file **server**, which at runtime responds to remote procedure calls from a client that supports the payroll interface, use the following commands.

```
% cc -c server.c
% fortran -c manager.for
% ld -o server server.o manager.o payroll_sstub.o
-lfor -lutil -lUfor -lm -lots -ldce -lpthreads -lmach -lc_r -lm
```

Instead of using these commands directly to build the server part of the application, you can use **make** to build the entire application (see *Section 12.3.8 on page 146*).

12.3.7 Client and Server Bindings

In order to make remote procedure calls, client applications must be bound to server applications. This is illustrated in the client program **print_pay.for** shown in *Section 12.3.4 on page 142*. The source code in the client program uses the default **[auto_handle]** binding, which is enabled by the following source code:

```
CCall remote procedure CALCULATE_PAY to
C       calculate the gross pay.


        CALL CALCULATE_PAY (CARDS, PAY)
```

When you run **make** (described in Building the Example (Makefile.unix)) or manually compile the application, a message is displayed about assuming **[auto_handle]**.

For more information about client and server bindings, see the chapter on basic DCE RPC runtime operations in the *OSF DCE Application Development Guide*.

12.3.8 Building the Example (Makefile.unix)

You can build the payroll example with **make** by using file **Makefile.unix**. Since the supplied Makefile has a **.unix** filename extension, you must use the **-f** option to the make command, as follows:

```
% make -f Makefile.unix
```

The contents of **Makefile.unix** follow.

```
#
#                       COPYRIGHT (C) 1993 BY
#                  DIGITAL EQUIPMENT CORPORATION, MAYNARD
#                   MASSACHUSETTS.  ALL RIGHTS RESERVED.
#
# THIS SOFTWARE IS 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.  THIS SOFTWARE OR ANY
# OTHER COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE
# TO ANY OTHER PERSON.  NO TITLE TO AND OWNERSHIP OF THE SOFTWARE
# IS HEREBY TRANSFERRED.
#
# THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT
# NOTICE AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL
# EQUIPMENT CORPORATION.
#
# DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS
# SOFTWARE ON EQUIPMENT THAT IS NOT SUPPLIED BY DIGITAL.
#
```

```
DCELIBS = $(LIBLOC) -ldce -lpthreads -lmach -lc_r -lm
FFLAGS = -c
FLIBS = -lfor -lutil -lUfor -lm -lots
FORTRAN = f77
I18NLIB =
IDL = idl
IDLFLAGS = -trace all -lang fortran -standard extended
LINKFLAGS =


# Default target - build client and server
all : print_pay server
@- cp /dev/null build.timestamp


# Target to build "local" (non-RPC) application in single image
local : local_print_pay
@- cp /dev/null buildl.timestamp


# Target to run local application
run_local :
FORT4=payroll.dat; export FORT4; local_print_pay


# Target to clean up non-source files
clean :
@- rm server server.o manager.o payroll_sstub.o
@- rm print_pay print_pay.o payroll_cstub.o
@- rm payroll.for payroll.for_h
@- rm build.timestamp
@- rm buildl.timestamp local_print_pay
@- rm server.log


server : server.o manager.o payroll_sstub.o
$(CC) $(LINKFLAGS) -o $@ server.o manager.o payroll_sstub.o \
$(FLIBS) $(DCELIBS)



print_pay : print_pay.o payroll_cstub.o
$(FORTRAN) $(LINKFLAGS) -o $@ print_pay.o payroll_cstub.o \
$(FLIBS) $(DCELIBS)


print_pay.o : print_pay.for payroll.for
$(FORTRAN) $(FFLAGS) -o $@ print_pay.for


payroll.for : payroll.idl
$(IDL) $(IDLFLAGS) payroll.idl


payroll_cstub.o : payroll.idl
$(IDL) $(IDLFLAGS) payroll.idl


server.o : server.c payroll.for_h
$(CC) $(CFLAGS) -o $@ server.c
```

```
payroll.for_h : payroll.idl
$(IDL) $(IDLFLAGS) payroll.idl


manager.o : manager.for payroll.for
$(FORTRAN) $(FFLAGS) -o $@ manager.for


payroll_sstub.obj : payroll.idl
$(IDL) $(IDLFLAGS) payroll.idl


local_print_pay : print_pay.o manager.o
$(FORTRAN) $(LINKFLAGS) -o $@ print_pay.o manager.o $(FLIBS) $(I18NLIB)
```

12.3.9 Running the Example

To run the sample application, perform the following steps:

1 Start the server process and run it as a background job:

```
% setenv RPC_DEFAULT_ENTRY
".:/FORTRAN_payroll_mynode"
% server &
```

2 Run the client:

```
% setenv FORT4 payroll.dat
% print_pay
```

The program displays the following output:

```
Jerry Harrison                  372
Tony Hardiman               294
Mary Flynn                          321
FORTRAN STOP
Deleting server process...
End of sample application
```

3 Bring the server process to the foreground and terminate it:

```
% fg
% CTRL/C
```

The output from building this application includes files **client** and **server**. You can use these executable programs in separate client and server processes.

12.4 Remote Procedure Calls Using FORTRAN — Reference

Remote Procedure Calls Using FORTRAN — Example contains a comprehensive example that introduces creating distributed applications with DIGITAL FORTRAN program units. This section goes beyond the example to provide reference information and explain general concepts about creating these distributed applications.

12.4.1 The FORTRAN Compiler Option

If you are generating stubs and include files for application code written in DIGITAL FORTRAN, you must specify it as the language of choice when you compile the application's IDL file. To specify the DIGITAL FORTRAN language, specify **-lang fortran**; the default value is **-lang c**.

In the remainder of this chapter, the phrase "FORTRAN option" refers to the IDL command that specifies DIGITAL FORTRAN. Examples of the IDL command and specification are presented in Compiling the Interface with the IDL Compiler.

Any client or server stub files that the FORTRAN option generates use the DIGITAL FORTRAN linkage conventions. This means that all parameters are passed by reference (see *Section 12.4.5.1 on page 153* for more information). In addition, all identifiers are converted to uppercase.

The FORTRAN option generates the file *filename*.**for**, which includes DIGITAL FORTRAN declarations of the constants and types declared in the IDL file. The **.for** file also includes, for each operation declared in the IDL file, a set of comments that describes the signature of the operation in DIGITAL FORTRAN terms.

In addition, the FORTRAN option generates the file *filename*.**for_h**. This file is used for generating the client and server stubs. It is also needed for generating DIGITAL FORTRAN stubs for any interface that imports this interface.

Consider the header option whose syntax is **-header**. If you specify both the FORTRAN option and the header option to the IDL compiler, the following rules govern the compiler's placement of the files *filename*.**for** and *filename*.**for_h**.

- If you specify a directory name in the header option, the compiler places the files in that directory. Otherwise, it places the files in the current default directory.
- If you specify a filename without an extension in the header option, the compiler uses that filename with the extensions **.for** and **.for_h.**
- If you specify a filename with an extension in the header option, the compiler uses that file extension instead of .**for_h**; however, the compiler does not change the extension of the .**for** file.

12.4.2 Restrictions on the Use of FORTRAN

This section discusses restrictions on distributed applications written in DIGITAL FORTRAN that make remote procedure calls. These restrictions are on interfaces and stubs, and on runtime operations.

- If an interface contains any arrays that have more than seven dimensions, the IDL compiler cannot produce output that is compatible with DIGITAL FORTRAN.
- If an interface contains two identifiers that differ only in the case of their characters, the IDL compiler might not be able to build stubs.

- The stubs generated for DIGITAL FORTRAN cannot call operations that use pipes.
- If the **transmit_as** or **represent_as** attributes have been applied to a character array type used to define the parameters of an operation, then DIGITAL FORTRAN cannot call that operation.
- If the **transmit_as** or **represent_as** attributes have been applied to an array type that, in turn, is the base type of an array type used to define the parameters of an operation, then DIGITAL FORTRAN cannot call that operation.
- If the **v1_array** attribute has been applied to any parameter of an operation, then DIGITAL FORTRAN cannot call that operation.
- DIGITAL FORTRAN does not allow the concurrent execution of two or more threads. In particular, if a server implements remote operations in DIGITAL FORTRAN, it must restrict the number of threads of server execution to 1. The following statement in file **server.c** (shown in *Section 12.3.5 on page 143*) specifies this restriction:

  ```
  rpc_server_listen((int)1, &st);
  ```

12.4.3 IDL Constant Declarations

A constant declaration either gives a name to an integer or string constant or gives a second name to a constant that has already been given a name. Examples of these declarations follow:

```
const long array_size = 100;
const char jsb = "Johann \"Sebastian' Bach";
const long a_size = array_size;
const boolean untruth = FALSE;
```

For all IDL constant declarations, equivalent PARAMETER statements are generated in the corresponding file *filename*.**for**. For example:

```
INTEGER*4 ARRAY_SIZE
PARAMETER (ARRAY_SIZE=100)


CHARACTER*(*) JSB
PARAMETER (JSB='Johann "Sebastian'' Bach')


INTEGER*4 A_SIZE
PARAMETER (A_SIZE=ARRAY_SIZE)


LOGICAL*1 UNTRUTH
PARAMETER (UNTRUTH=.FALSE.)
```

All integer constants are declared as INTEGER*4.

All **void *** constants are ignored.

A nonprinting character that appears within a character or string constant is replaced by a question mark (?).

12.4.4 Type Mapping

An IDL type that is a synonym for another type is presented to DIGITAL FORTRAN as the type for which the synonym is defined. For example, suppose that the IDL file contains the following statement:

```
typedef foo bar;
```

Then, all instances of IDL type `bar` are presented to DIGITAL FORTRAN as of type `foo`.

The next table describes the mappings from IDL types to DIGITAL FORTRAN types.

Table 12-3: Mappings for IDL Types

IDL Data Type	FORTRAN Data Type	Comments
arrays		See notes 8 and 9
boolean	LOGICAL*1	
byte	BYTE	
char	CHARACTER	
context handle	INTEGER*4	
double	REAL*8	See note 3
enum	INTEGER*4	
error_status_t	INTEGER*4	See note 4
float	REAL*4	
handle_t	HANDLE_T	See ***Section 12.4.7 on page 154***
hyper	IDL_HYPER_INT	See ***Section 12.4.7 on page 154***
ISO_MULTI_LINGUAL	ISO_MULTI_LINGUAL	See ***Section 12.4.7 on page 154***
ISO_UCS	ISO_UCS	See ***Section 12.4.7 on page 154***
long	INTEGER*4	
pipe		No mapping
pointer	INTEGER*4	See note 10
short	INTEGER*2	
small	INTEGER*2	See note 1
struct	STRUCTURE	See notes 5 and 6
union	UNION	See note 7
unsigned hyper	IDL_UHYPER_INT	See ***Section 12.4.7 on page 154***
unsigned long	INTEGER*4	See note 2
unsigned short	INTEGER*4	See note 1
unsigned small	INTEGER*2	See note 1

Notes

1 For these IDL data types, the DIGITAL FORTRAN data type is chosen because it can represent all possible values of the IDL type. Note that, in each case, there are values of the DIGITAL FORTRAN type, which cannot be represented in the IDL type. You must not attempt to pass such values in parameters. The RPC runtime code does not perform range checking.

2 Because some values that can be represented in an IDL data type cannot be represented correctly in the DIGITAL FORTRAN data type, the IDL compiler issues a warning.

3 You must compile DIGITAL FORTRAN code that uses this data type and specify the **cc-cmd** *'command_line'* compiler option.

4 Status code mapping will occur where necessary.

5 For any structure type in the IDL file that is not defined through a typedef statement, the IDL compiler generates the name of the DIGITAL FORTRAN structure. To determine what name was generated, look at *filename*.**for**.

6 The semantics of conformant structures cannot be represented in DIGITAL FORTRAN. In the definition of such a structure in *filename*.**for**, a placeholder for the conformant array field is specified as a one-dimensional array with one element. If the first lower bound of the conformant array is fixed, this value is used as the lower and upper bounds of the placeholder. If the first lower bound of the array is not fixed and if the first upper bound of the conformant array is fixed, the upper bound is used as the lower and upper bounds of the placeholder. Otherwise, the lower and upper bounds of the placeholder are zero.

7 Note that IDL encapsulated union types and nonencapsulated union types are represented as DIGITAL FORTRAN structures containing unions.

8 IDL array types are converted to arrays of a nonarray base type.

9 Arrays that do not have a specified lower bound have a lower bound of zero. Consider the following two statements in an IDL file:

```
double d[10][20];
short e[2..4][3..6];
```

The statements map into the following DIGITAL FORTRAN constructs.

```
REAL*8 D(0:9,0:19)
INTEGER*2 E(2:4,3:6)
```

10 The size of the pointer depends on the platform. It is INTEGER*8 for Tru64 UNIX systems.

12.4.5 Operations

Operations can pass parameters and return function results. This section explains these two topics.

12.4.5.1 Parameter Passing by Reference

The following rules explain the mapping between IDL parameters and DIGITAL FORTRAN parameters.

- If the IDL parameter contains an asterisk (*) and does not have a **[ptr]** or **[unique]** attribute, this signifies a parameter of the indicated type passed by reference. The DIGITAL FORTRAN parameter is of the same type.
- If the IDL parameter contains an asterisk and does have a **[ptr]** or **[unique]** attribute, the DIGITAL FORTRAN parameter is a pointer.
- If the IDL parameter is an array and has the **[ptr]** or **[unique]** attribute, the DIGITAL FORTRAN parameter is a pointer.
- If none of the preceding cases is true, then the DIGITAL FORTRAN parameter is of the same type as the IDL parameter.

12.4.5.2 Function Results

The only possible function result types in DIGITAL FORTRAN are scalars and CHARACTER*_n_. The mappings from IDL to DIGITAL FORTRAN never produce CHARACTER*_n_, where _n_ is greater than 1.

IDL hyper integers are not scalars in terms of function results, but IDL pointers are treated as scalars because they are mapped to INTEGER*8.

For an operation that has a result type that is not allowed by DIGITAL FORTRAN, the stubs treat the operation result as an extra **[out]** parameter added to the end of the parameter list.

If the type of an operation is not **void**, you must state the type of the function result in DIGITAL FORTRAN.

12.4.6 Include Files

Usually, a DIGITAL FORTRAN routine that is part of an RPC client or manager for the interface defined in *filename*.**idl** must include the following files:

- filename.**for**
- **nbase.for**
- The .**for** files for any imported interfaces

Program units **print_pay.for** and **manager.for** (containing subroutine subprogram CALCULATE_PAY) in the example of a distributed payroll application do not include **nbase.for** because the units contain none of the IDL data types. Otherwise, the program units would include **nbase.for**. Furthermore, these units could safely include **nbase.for** even though it is unnecessary in the example.

12.4.7 The nbase.for File

The file **/usr/include/dce/nbase.for** declares standard data types used in mapping IDL to DIGITAL FORTRAN. The declarations are shown in *Table 12-4*.

Table 12-4: Standard Declarations

IDL Data Type	FORTRAN Declaration
hyper	STRUCTURE /IDL_HYPER_INT/ INTEGER*4 LOW INTEGER*4 HIGH END STRUCTURE
unsigned hyper	STRUCTURE /IDL_UHYPER_INT/ INTEGER*4 LOW INTEGER*4 HIGH END STRUCTURE
handle_t	STRUCTURE /HANDLE_T/ INTEGER*4 OPAQUE_HANDLE END STRUCTURE
ISO_MULTI_LINGUAL	STRUCTURE / ISO_MULTI_LINGUAL/ BYTE ROW BYTE COLUMN END STRUCTURE
ISO_UCS	STRUCTURE /ISO_UCS/ BYTE GROUP BYTE PLANE BYTE ROW BYTE_COLUMN END STRUCTURE

NOTE: For IDL data type handle_t, the size of pointers is platform specific: on OpenVMS systems, pointers are INTEGER*4 and on Tru64 UNIX systems, pointers are INTEGER*8.

12.4.8 IDL Attributes

This section describes IDL attributes that apply to RPC applications containing DIGITAL FORTRAN modules.

12.4.8.1 Binding Handle Callout

The Binding Handle Callout feature lets you specify a routine that is automatically called from an IDL-generated client stub routine, in order to modify the binding handle.

You can typically use this feature to augment the binding handle with security context, for example, so that authenticated RPC calls are used between client and server.

This feature is particularly targeted at clients which use automatic binding via the **auto_handle** ACF attribute. For automatic binding, it is the client stub rather than the client application code which obtains a server binding handle. The binding callout feature lets you modify binding handles obtained by the client stub. Without this feature, you cannot modify the binding handles before the client stub attempts to initiate a remote procedure call to the selected server.

12.4.8.2 ACF file

To select the binding handle callout feature, create an ACF file for the interface (if necessary) and place the **binding_callout** attribute on the interface. An example follows:

```
[auto_handle, binding_callout(my_bh_callout)] interface bindcall
{
}
```

The **binding_callout** attribute has the following general form:

```
[binding_callout(identifier)]
```

You can specify the **binding_callout** only once per interface; it applies to all operations in that interface.

12.4.8.3 Generated header file

The IDL-generated header file for the interface contains a function prototype for the binding callout routine. In the previous example, bindcall.h contains a declaration similar to the following declaration:

```
void my_bh_callout(
    rpc_binding_handle_t *p_binding,
    rpc_if_handle_t interface_handle,
    error_status_t *p_st
);
```

12.4.8.4 Generated client stub

Each client stub routine in the IDL-generated client stub module calls the binding callout routine just before initiating the remote procedure call to the server. In the previous example, each client stub routine contains a call to my_bh_callout and passes the three arguments that are described in the following section.

12.4.8.5 Binding callout routine

The arguments to the binding callout routine are:

- Input/Output

  ```
  rpc_binding_handle_t *p_binding
  ```

  A pointer to a server binding handle for the remote procedure call. Generally, the binding callout routine will augment this binding handle with additional context, such as for security.

- Input

```
rpc_if_handle_t interface_handle
```

The interface handle used to resolve a partial binding or for the binding callout routine to distinguish interfaces.

- Output

```
error_status_t *p_st
```

An error status code returned by the binding callout routine.

12.4.8.6 Error handling

A binding callout routine returns error_status_ok when it successfully modifies the binding handle or decides that no action is necessary. This causes the client stub to initiate the remote procedure call.

When the binding callout routine returns an error status, the client stub will not initiate a remote procedure call. If **auto_handle** is being used, the client stub will attempt to locate another server of the interface and once again call the binding callout routine. Otherwise, it will execute its normal error handling logic.

A binding callout routine for a client using **auto_handle** can return rpc_s_no_more_bindings to prevent the client stub from trying to locate another server. The client stub will then execute its normal error handling logic.

By default, a client stub handles an error condition by raising an exception. If a binding callout routine returns an rpc_s_ status code, the client stub raises the matching rpc_x_ exception. If a binding callout routine returns any other error status, it is usually raised as an “unknown status” exception.

For an operation containing a **comm_status** parameter, the client stub handles an error condition by returning the error status value in the [comm_status] parameter. A binding callout routine can return any error status value to the client application code if the IDL operations are specified with **comm_status** parameters.

A binding callout routine can raise a user-defined exception rather than return a status code if it prefers to report application-specific error conditions back to the client application code via exceptions.

12.4.8.7 Predefined binding callout routine

There is one predefined binding callout routine in the DCE library which may be suitable for some applications. To select this routine, specify a **binding_callout(rpc_ss_bind_authn_client)** ACF attribute.

rpc_ss_bind_authn_client matches the function prototype in the previous section, Generated Header File. It authenticates the client identity to the server, thereby allowing for one-way authentication. In other words, the client does not care which server principal receives the remote procedure call request, but the server verifies that the client is who the client claims to be.

rpc_ss_bind_authn_client does the following:

- Calls rpc_ep_resolve_binding**( )** to resolve the binding handle if it is not fully bound (For example, for **auto_handle**).
- Calls rpc_mgmt_inq_server_princ_name**( )** to obtain the server identity (principal name).
- Calls rpc_binding_set_auth_info**( )** with all default values except the server principal name obtained in the previous call.
- If any of these calls returns an error status, places the error status in the *p_st argument and rpc_ss_bind_authn_client returns.

12.4.8.8 The transmit_as Attribute

The presented type must be expressible in DIGITAL FORTRAN. Because addresses are involved, the routines used for data conversion cannot be written in DIGITAL FORTRAN.

12.4.8.9 The string Attribute

A DIGITAL FORTRAN data item corresponding to an IDL string contains the number of characters specified for the IDL string. Because IDL strings are usually terminated with a null byte, the following transmission rules apply:

- If a DIGITAL FORTRAN routine contains data for transmission, and a null byte appears before the last character of the DIGITAL FORTRAN data item, then the characters up to and including the null byte are transmitted.
- If a DIGITAL FORTRAN routine contains data for transmission, and a null byte does not appear before the last character of the DIGITAL FORTRAN data item, then all the characters of the data item except the last are transmitted, followed by a null character.
- If data is transmitted to a DIGITAL FORTRAN routine, then the DIGITAL FORTRAN data item receives a null terminated string. If the DIGITAL FORTRAN data item contains more characters than the string, then the additional characters are not affected.

An IDL operation can have a conformant string parameter. Such a parameter is presented to DIGITAL FORTRAN as type CHARACTER*(*). If the base type of the string consists of *w* bytes and the string consists of *n* characters, then the parameter size is *n***w.* The maximum parameter size supported is 65535.

A conformant string field of a structure will have type CHARACTER**w,* where *w* is the number of bytes in the base type of the string.

In all other cases where a string is not the target of a pointer, the IDL file specifies the string. Such a string is presented to DIGITAL FORTRAN as CHARACTER**s*, where *s* is the product of the string length and the number of bytes in the base type of the string. Furthermore, *s* must be between 1 and 65535 inclusive.

12.4.8.10 The context_handle Attribute

A context handle rundown routine cannot be written in DIGITAL FORTRAN because the routine must handle address information.

12.4.8.11 The Array Attributes on [ref] Pointer Parameters

A **[ref]** pointer parameter that has array attributes attached to it is presented to DIGITAL FORTRAN as the equivalent array.

12.4.9 ACF Attributes

The following items can occur in an Attribute Configuration File (ACF). They require special consideration when you are using DIGITAL FORTRAN.

12.4.9.1 The implicit_handle ACF Attribute

You must supply a COMMON block whose name is the name given in the implicit handle clause. This COMMON block must contain the binding handle as its only data item.

For example, suppose an ACF contains the following interface attribute:

```
[implicit_handle(handle_t i_h)]
```

Then, any DIGITAL FORTRAN routine that calls an operation which uses the implicit binding must include statements with the following form:

```
RECORD /HANDLE_T/ BINDING_HANDLE
COMMON /I_H/      BINDING_HANDLE
```

12.4.9.2 The represent_as ACF Attribute

The local type must be expressible in DIGITAL FORTRAN.

Because addresses are involved, you cannot write the data conversion routines in DIGITAL FORTRAN.

A type name in a **represent_as** attribute that does not occur in the interface definition and is not an IDL base type is assumed to be a STRUCTURE type.

Suppose that the **represent_as** type is not an IDL base type or a type defined in your IDL source. Then, you must supply a **.h** file whose unextended name is given in an include statement in the ACF. (An unextended name is a filename without the file extension that follows the final dot (.) in the name. For example, the unextended filename for file **example.h** is **example**.) This file must include a definition of the local type in C syntax. You will need a *filename*.**for** file containing a DIGITAL FORTRAN definition of the local type. Entegrity recommends that you assign this file the same unextended name.

CHAPTER 13

Example Programs

13

13.1 Overview of Remote Procedure Call Programs

Several example programs are supplied with the Application Developer's Kit subset. These programs are located in directories under **/usr/examples/dce**. This chapter provides information about the example programs provided with Gradient DCE for Tru64 UNIX. Each example program also has an online README file located in the same directory as the program. The next table shows the different features of each example program.

Table 13-1: Features of Example Programs

Example Program	Description
RPC Test Program #1	Server does not register endpoints; binding information not exported to namespace.
RPC Test Program #2	Server registers endpoints; binding information exported to namespace; uses security.
RPC Test Program #3	Server registers endpoints; binding information not exported to namespace.
Book Program	Server registers endpoints; binding information is exported to namespace; uses mutex locks and security.
Time Operations Program	Uses all DCE services, including serviceability, security and threads.
Phonebook Program	Uses RPC and the name service.
Echo Program	Demonstrates how a distributed application can secure itself using the GSSAPI security interface.
Time Provider Programs	Illustrate how to structure and use programs for external time providers.
Serviceability Program	Demonstrates the use of the serviceability API.
Generic Application	Demonstrates ACL management, serviceability code, security setup, and signal handling.
Object Oriented Programs	Demonstrate the use of C++ idl extensions.

Copy the example files to another area before you attempt to build them. You also may want to open two separate windows for the client and server processes.

The following sections describe the example programs.

13.2 RPC Test Program #1

RPC Test Program #1 is a simple client/server program that makes minimal use of the DCE services. The server does not register transport endpoints with the DCE daemon, and no binding information is exported to the directory service. The server binding information has to be transferred to the client manually by the user.

To build this example program, enter the following commands:

```
% cp /usr/examples/dce/rpc/test1/* .
% make -f makefile.test1
```

After the build is completed, start the server with the following command syntax:

```
% test1d [protseq]
```

The server reports binding information for each of the various protocol sequences that are available and both displays the information on the terminal screen and writes it to a file called binding.dat. This binding information consists of three elements: a protocol sequence, a network address, and a transport endpoint. For example, the server might report the following binding information:

```
ncacn_ip_tcp 66.0.0.7 4344
```

If you want the server to use a specific protocol sequence, you can include that as an argument in the server startup command. For example:

```
% test1d ncacn_ip_tcp
```

This command causes the server to use that protocol sequence only. The protocol sequences currently supported include **ncacn_ip_tcp** (connection protocol) and **ncadg_ip_udp** (datagram protocol).

Once the server is running, you can run the client on the same host or on any other host in the network. To run the client, you must provide the server binding information reported by the server. For example, you can run the client with the following command syntax:

```
% test1 protseq networkaddr endpoint [number of passes] [calls per pass]
```

For example:

```
% test1 ncacn_ip_tcp 66.0.0.7 4344
% test1 ncacn_ip_tcp hostname 4344
```

The client makes a number of remote procedure calls, each of which causes a simple arithmetic function to execute. After making a sequence of calls, the client reports the average elapsed time for the calls to complete. By default, the client makes 10 passes with 100 calls per pass. You can specify the number of passes and the number of calls per pass by adding two additional arguments to the client startup command. For example, the following command instructs the client to make 5 passes, with 1000 calls per pass:

```
% test1 ncacn_ip_tcp 66.0.0.7 4344 5 1000
```

Because of the granularity of the clock on most systems, the average time per call will not be very accurate unless you set the number of calls per pass to a relatively high number (at least 1000).

The client can be run as many times as desired, as long as the server is still running. If you want to clean up the directory for this test so that you can build it again, enter the following command:

```
% make -f makefile.test1 clean
```

13.3 RPC Test Program #2

RPC Test Program #2 is a simple client/server program that makes more use of the DCE services than RPC Test Program #1. In this program, the server registers transport endpoints with the DCE daemon and exports binding information to the directory service. The client uses the auto-handle mechanism to import server binding information.

To build this example program, enter the following commands:

```
% cp /usr/examples/dce/rpc/test2/* .
```

```
% make -f makefile.test2
```

Because this program exports and imports an entry to the global namespace (.:), you must perform a **dce_login** operation as **cell_admin** or some other privileged principal before you start the server process. Start the server with the following command:

```
% test2d
```

Once the server is running, you can run the client on the same host or on any other host in the network that is configured to run in the same cell as the server host. Before running the client, you must define an environment variable on the client system that can be used to locate the server binding information in the namespace during the auto-handle process:

```
% setenv RPC_DEFAULT_ENTRY /.:/test2_server
```

After you define the environment variable, run the client with the following command syntax:

```
% test2 [passes][calls per pass]
```

The client imports server binding information from the namespace. It makes a number of remote procedure calls, each of which causes a simple arithmetic function to execute. After making a sequence of calls, the client reports the average elapsed time for the calls to complete. By default, the client makes 10 passes with 100 calls per pass. You can specify the number of passes and the number of calls per pass by adding two arguments to the client startup command. For example, the following command instructs the client to make 5 passes, with 1000 calls per pass:

```
% test2 5 1000
```

Because of the granularity of the clock on most systems, the average time per call will not be very accurate unless you set the number of calls per pass relatively high (at least 1000).

The client can be run as many times as desired, as long as the server is still running. If you want to clean up the directory for this test so that you can build it again, enter the following command:

```
% make -f makefile.test2 clean
```

13.4 RPC Test Program #3

RPC Test Program #3 is a simple client/server program that makes minimal use of the DCE services. The server registers transport endpoints with the DCE daemon, but no binding information is exported to the directory service.

To build this example program, enter the following commands:

```
% cp /usr/examples/dce/rpc/test3/* .

% make _f makefile.test3
```

After the build is completed, ensure that **dced** is running, and then start the server with the following command:

```
% test3d [protseq]
```

The server reports binding information for each of the various protocol sequences that are available and both displays the information on the terminal screen and writes it to a file called **binding.dat**. This binding information consists of two elements: a protocol sequence and a network address. For example, the server might report the following binding information:

```
ncacn_ip_tcp 66.0.0.7
```

If you want the server to use some specific protocol sequence, you can include that as an argument in the server startup command. For example:

```
% test3d ncacn_ip_tcp
```

This command causes the server to use that protocol sequence only. The protocol sequences currently supported include **ncacn_ip_tcp** (connection protocol) and **ncadg_ip_udp** (datagram protocol).

Once the server is running, you can run the client on the same host, or on any other host in the network. To run the client, you must provide the server binding information reported by the server. For example, you can run the client with the following command syntax:

```
% test3 protseq hostaddr [passes] [calls per pass]
```

For example:

```
% test3 ncacn_ip_tcp 66.0.0.7
```

or

```
% test3 ncacn_ip_tcp hostname
```

The client makes a number of remote procedure calls, each of which causes a simple arithmetic function to execute. After making a sequence of calls, the client reports the average elapsed time for the calls to complete. By default, the client makes 10 passes with 100 calls per pass. You can specify the number of passes and the number of calls per pass by adding two additional arguments to the client startup command. For example, the following command instructs the client to make 5 passes, with 1000 calls per pass:

```
% test3 ncacn_ip_tcp 66.0.0.7 5 1000
```

Because of the granularity of the clock on most systems, the average time per call will not be very accurate unless you set the number of calls per pass to a relatively high number (at least 1000).

The client can be run as many times as desired, as long as the server is still running. If you want to clean up the directory for this test so that you can build it again, enter the following command:

```
% make _f makefile.test3 clean
```

13.5 Book Distributed Calendar Program

The Book distributed calendar program (**book**) is a fairly sophisticated client/server application that uses of a number of DCE services. The program registers transport endpoints with the DCE daemon and exports server binding information to the directory service. It also demonstrates some minimal use of mutex locks to protect resources on the server from access by multiple call threads.

To build this example program, enter the following commands:

```
% cp /usr/examples/dce/rpc/book/* .
```

```
% make -f makefile.book
```

After the build is completed, log in as **root**, perform a **dce_login** operation, and start the server with the following command:

```
% bookd [-d][-v][bookname]
```

The server calls a useful set of initialization routines from the DCE library. The first call, **rpc_server_init( )**, forks a process to run the server and initializes the RPC runtime with the appropriate parameters. The program does this before any other calls are made by the server to the RPC runtime and before any other threads calls are made (because thread context is not guaranteed to be preserved across a fork). After performing other initialization functions (this program initializes a global mutex), the program makes a second call to **rpc_server_detach( )**. This call releases the terminal associated with the parent process, after which the parent process is free to exit. The server then starts listening for client requests.

The server takes three optional command arguments that affect the initialization sequence, as shown in *Table 13-2*.

Table 13-2: Options for Starting the Book Distributed Calendar Program

Argument	Description
-d	(Debug mode) Do not fork a child process (run the server in the parent). Default: No Debug mode.
-v	(Verbose mode) Display informational messages during initialization. Default: No Verbose mode.
bookname	Calendar name to be used. Default: *login_name*.**book**.

Once the server is running, you can run the client on the same host, or on any other host in the network that is configured to run in the same cell as the server host. Start the client with the following command:

```
% book [bookname]
```

The client imports server binding information from the directory service, and causes the server to update the calendar file for the account in which the client is running. The client has a help facility that lists the commands that you can execute to modify the calendar database on the server.

You can execute the client as many times as desired, as long as the server is still running. To clean up the directory for this application so you can build it again, enter this command:

```
% make -f makefile.book clean
```

13.6 The Time Operations Sample Application

The **timop_svc** program exercises the basic DCE technologies: threads, RPC, security, directory, time and serviceability. Its detailed description is in the README file the **/usr/examples/dce/svc/timop_svc** directory.

13.6.1 Overview

The two parts of **timop** are a client and a server, implemented by the **timop_svc_client** and **timop_svc_server** processes. The server manages a single operation, getting the span of time used in calculating the factorial of a random number specified by the client. The client spawns several threads that make parallel RPC service calls to designated servers. The client prints the name, invocation order, and time span that each server reports, and the number for which the server calculated the factorial. It also prints out a total time span that encompasses all the job events at the servers and the sum of the random numbers.

The transport provider is UDP. Authentication and integrity-secure RPC ensure data communication. Named-based authorization (not ACLs) is employed. Clients and servers use different physical clocks that are in agreement with one another because they are synchronized by the time service. All times and time calculations are in UTC, not local civil time; permitting clients and servers to operate in different time zones.

Because **timop** uses the security service, the timop clients and servers must run as security principals, but with only minimum use of security. The **timop_client** runs as a principal named **/.../mycell/tclient** and **timop_server** as a principal named **/.../mycell/tserver**. These names can be changed to suit your environment by modifying **timop_svc_aux.h** file.

Additional information on serviceability can be found in the *OSF DCE Application Development Guide - Core Components* volume. See also the **log.8dce** reference page (about the dcecp log object, through which the DCE components' serviceability routes and settings are managed) in the *OSF DCE Command Reference*.

13.6.2 Building timop_svc

Before building **timop_svc**, make sure that the DCE Application Development Environment (which includes the IDL compiler) is installed. Next, read the comments of the Makefile, and remove comment flags and options as appropriate to your platform.

To build timop_svc, enter the following command.

```
% make -f Makefile.timop_svc
```

13.6.3 Setting Up to Run timop_svc

Before you can run **timop_svc**, you must first set up your DCE cell with the security registry and namespace information necessary for the program and for its client and server principal entities. You must also set up an authentication key table file on each machine on which you intend to run the server. To do so, follow these steps:

1. Add the client and server principals to the Security registry.
2. Create a keyfile to be used by the server.
3. Create a CDS namespace entry, to which the server exports its binding information, and from which the clients import it.
4. Set up the correct permissions on the namespace entry so that the server can write to it correctly.

Included with the example's files are a pair of dcecp scripts that automatically perform (or undo) all of the above steps, except the second step (creating the keytab file). Each script also logs in to the cell as cell_admin as its first operation. The login operation uses the default cell password (-dce-).

The following examples show how to run the setup script:

```
% ./timop_svc_setup.dcecp /.:/ts_entry

principal create {tsserver tsclient}
group add none -member tsserver
group add none -member tsclient
organization add none -member tsserver
organization add none -member tsclient
account create tsserver -group none -organization none \
-password qwerty -mypwd -dce-
account create tsclient -group none -organization none \
-password xyzzy -mypwd -dce-
Adding CDS entries.
```

Once the setup script has been run, you should log in as the Cell Administrator using dce_login and run rgy_edit to set up the server's keyfile:

```
% dce_login cell_admin -dce-
% rgy_edit
Current site is: registry server at /.../your_cell/subsys/dce/sec/master
rgy_edit=> ktadd -p tsserver -pw qwerty -f /tmp/tskeyfile
rgy_edit=> quit
bye.
```

You have now finished the **timop_svc** setup.

For more information about **rgy_edit**, see the *OSF DCE Administration Guide - Core Components and the OSF DCE Command Reference.*

The name of the server's keyfile, **/tmp/tskeyfile**, is specified by the value of the KEYFILE constant in the **timop_svc_server.h** file; the name you give to the **ktadd** subcommand must be identical to the value of this constant.

To undo the setup, run the **unsetup** script, as follows:

```
% ./timop_svc_unsetup.dcecp /.:/ts_entry
principal delete {tsserver}
principal delete {tsclient}
account delete tsserver
Registry object not found
account delete tsclient
Registry object not found
Deleting CDS entries.
```

13.6.4 timop_svc Message Catalog

The DCE Serviceability API uses XPG4 message catalogs to store and retrieve message text. The catalogs are generated by the DCE **sams** utility. The catalogs should be installed in their correct platform-specific location. For example:

/usr/lib/nls/msg/*LANG*

However, if the serviceability routines cannot find a catalog there, they default to their respective current working directory. If they cannot find the catalogs there either, they retrieve messages from the in-memory table, if one has been defined by the application. See the Serviceability chapter in the *OSF DCE Application Development Guide* for details. Thus you should be able to run **timop_svc** successfully without doing any extra message catalog installation.

13.6.5 Running the timop_svc Server

To run **timop_svc**, you must first start the server and invoke one or more clients to perform the **timop_svc** operation. An example of how to do this follows.

On the machine on which you want to run the server, enter this command:

```
% ./timop_svc_server -e1 /.:/ts_entry
```

NOTE: You should start the server in the background, in a window different from the one in which you intend to run the client, or on a separate terminal.

The **/.:/ts_entry** is the server's name in the namespace. It is the name of the CDS entry to which it exports its bindings, and therefore is the name by which it is known to clients. The entry was set up when you ran the **dcecp** setup script earlier; it can have any name you choose.

The **-e1** specifies the object UUID that the server should export and register its bindings with. Two object UUIDs are available, specified to the server as **-e1** or **-e2**. Having two UUIDs allows you to have two servers running at the same time (and even on the same machine). Clients can bind to the server they choose simply by specifying the correct object number in the client command line (as will be seen below). Even when two **timop_svc** servers export to different name entries, if the servers are active at the same time, their exported

partial bindings will be identical if they are running on the same machine; requiring that the bindings be exported and imported with an object UUID specified prevents different server instances from getting mixed up.

The server displays a series of messages, most of them output through the Serviceability API. For more information about these messages and how to control them, see the sample output in the README file. At the end of all this preliminary activity, the server displays a "ready" message.

13.6.6 Running the timop_svc Client

After you have invoked the server, wait until you get a message similar to this one:

```
1994-05-26-19:36:32.915+00:00I----- ./timop_svc_server NOTICE tsv server
0x7aff3f20
Server /.:/ts_entry (object 1) ready...
```

(This is the serviceability form of the "Server ready" message displayed by the **timop** server.)

You can now invoke the client (either in the same window, if you ran the server in the background, or in a different window). To get rid of your **tsserver** identity when invoking the client from the same window, enter:

```
% exit
```

Next, log in as the **tsclient** principal and then start the **timop_svc_client** program. Enter:

```
% dce_login tsclient xyzzy
% timop_svc_client -o1 /.:/ts_entry
```

The **-o1** specifies that the client is to import the bindings registered with object UUID 1, which is the object the server exported to. If the server had specified **-e2**, then the client would have had to specify **-o2**. If two servers were active and each had exported to a different object, clients could specify either object (or both) to import.

If all has gone well, the **timop_svc** client now begins printing out results continuously until you stop it. (See *Section 13.6.8 on page 168* for details on how to do this.)

On multiple machines in the same cell, you can try something like the following:

```
% timop_svc_server -e1 /.:/ts_entry # on machine A
% timop_svc_server -e2 /.:/xs_entry # on machine B
% timop_svc_client /.:/ts_entry /.:/xs_entry /.:/ys_entry # on machine D
% timop_svc_client /.:/ys_entry /.:/xs_entry /.:/ts_entry # on machine E
```

To do this, you must first set up **xs_entry** and **ys_entry** names in CDS by specifying these additional names to the **timop_svc_setup.dcecp** script.

13.6.7 Sample Server Output

Following is an example of the kind of server output you can expect to see if you invoke the **timop_svc** server with full debugging enabled, and with serviceability NOTICE type messages routed to standard error.

In general, the first groups of messages are output as a result of straightforward test calls to various routines; the later messages contain authentic information being output via the serviceability interface. As explained above, once this message appears, the server waits, and the client (or clients) can then be started.

```
1994-05-26-19:36:32.915+00:00I----- ./timop_svc_server NOTICE tsv server
0x7aff3f20
Server /.:/ts_entry (object 1) ready...
```

Once a client is started, the server resumes its messaging activity; the amount of activity is determined by the debug level you specify (the default is no debug messaging) and the routing you have set. In the preceding example, the "Server ... ready" message is about the 16th from the last; the subsequent messages represent a sample of what happens once a client has become active.

See the **timop_svc** source code (which is fully commented) for details of which serviceability routines are called.

Refer to *Section 13.6.9 on page 168* and *Section 13.6.10 on page 169* for more information on how to specify various aspects of **timop_svc**'s behavior.

For more information on Serviceability functionality, see the serviceability chapter in the *OSF DCE Application Development Guide - Core Components* volume.

See the README file for examples of server and client output.

13.6.8 Stopping timop_svc

You must kill clients and servers by hand, either by using the interrupt key or with a combination of the **ps** and **kill** commands. Doing so leaves server binding information in the endpoint map and namespace, which is normal for persistent servers. The information can be removed afterwards by running the **timop_svc_unsetup.dcecp** script.

13.6.9 timop_svc Server Options

The **timop_svc** server is invoked as follows:

```
timop_svc_server [-wsvc_route [-wsvc_route ... ]] \
[-d"dbg_route" [-d"dbg_route" ... ]] \
[-D"dbg_level" [-D"dbg_level" ... ]] \
[-f] -enr entry_name
```

where:

-w*svc_route* (optional, one or more)	Specifies a serviceability routing.
-d"*dbg_route***"** (optional, one or more)	Specifies a serviceability debug routing.
-D"dbg_level**"** (optional, one or more)	Specifies a serviceability debug level.
-f (optional)	Causes the serviceability filter to be installed.
-e1 or **-e2**	Specifies the object entry this server instance is using for export.
entry_name	Specifies the name of the entry to which this server instance should export.

For more information on Serviceability functionality, see the serviceability chapter in the *OSF DCE Application Development Guide - Core Components* volume.

13.6.10 timop_svc Client Options

The **timop_svc** client is invoked as follows:

timop_svc_client _o*nr* [**-o***nr* ...] \
server_entry [*server_entry* ...] \
[**-D"***dbg_level***"** [**-D"***dbg_level***"** ...]] \
[**-d"***dbg_route***"** [**-d"***dbg_route***"** ...]] \
[**-w***svc_route*[**-w***svc_route* ...]] [**-l**] [**-C**] [**-R**] [**-f**]

or:

timop_svc_client -o*nr* **-b"***string_binding***"** \
[**-D"***dbg_level***"** [**-D"***dbg_level***"** ...]] \
[**-d"***dbg_route***"** [**-d"***dbg_route***"** ...]] \
[**-w***svc_route*[**-w***svc_route* ...]] [**-l]** [**-C**] [**-R**] [**-f**]

where:

-o1 or **-o2**	Specifies the server object to bind to. You can specify up to 2 objects. [NOTE: This limit, and the values of the object UUIDs, are defined in **timop_svc_aux.h**. You can increase the number of objects allowed by altering the contents of this file and rebuilding **timop_svc**.]
server_entry	Specifies the name of the server entry to bind to; You can specify up to 10 *server_entrys*. If you specify multiple servers and objects, the list of servers and the list of objects must ordinally match. [NOTE: This limit is specified in **timop_svc_client.h**.]
string_binding	Specifies a complete binding to use to make direct contact with the server. Multiple servers cannot be specified with this option, and specifying a *server_entry* with it is an error.
9_D"*dbg_level***"** (optional, one or more)	Specifies a serviceability debug level. For example: **-D"tsv:tsv_s_server.5,tsv_s_refmon.9"** or: **-D"tsv:*.9"** See *Section 13.6.12 on page 170* for the significance of the various available levels for **timop_svc**.
-d"*dbg_route***"** (optional, one or more)	Specifies a serviceability debug routing. For example: **-d"tsv:tsv_s_server.5:TEXTFILE:***pathname***"** or: **-d"tsv:*.8:STDERR:"**
-w*svc_route* (optional, one or more)	Specifies a remote serviceability routing.
-l	Specifies that the serviceability subcomponents be listed.
-C	Specifies that all registered serviceability components be listed.
-R	Specifies that the serviceability routings be listed.
-f	Specifies that the remote serviceability filter routine be toggled.

For more information on Serviceability functionality, see the *OSF DCE Application Development Guide - Core Components*.

13.6.11 timop_svc Principal And Keytab Names

tsserver	Server principal name [defined in **timop_svc_aux.h**]
tsclient	Client principal name [defined in **timop_svc_aux.h**]
/tmp/tskeyfile	keytab pathname [defined in **timop_svc_aux.h** and **timop_svc_server.h**]

13.6.12 timop_svc Debug Message Levels

You can set to nine different debug levels (by means of the **_D** switch in the server or client command line; see ***Section 13.6.10 on page 169***). *Table 13-3* shows the debug level significance in **timop_svc**:

Table 13-3: timeop_svc Debug Message Levels

Level	Meaning
1	Used for test messages in “server” subcomponent.
2	Used for test messages in “server” subcomponent.
3	Used for test messages in “server” subcomponent.
4	Used for test messages in “server” subcomponent.
5	In “server”, “manager”, and “refmon” subcomponents,enables messages that are written at each DCE library call. In “manager” and “refmon”, also enables messages whenever local subroutines are entered or exited. In “refmon”, also enables messages describing values about to be returned by local subroutines.
6	Used for test messages in “server” subcomponent.
7	In “remote” subcomponent, enables messages that are written whenever a remote serviceability routine is entered, as well as messages that are written at each DCE library call.
8	Used for test messages in “server” subcomponent.
9	Used for test messages in “server” subcomponent.

For more information on Serviceability functionality, see the *OSF DCE Application Development Guide - Core Components.*

13.7 Microsoft RPC Phonebook Program

This section describes how to build and run a phonebook application called **phnbk**. Company employees use the **phnbk** client program to look up employee contact information that resides with the **phnbk** server.

The **phnbk** application is included with Gradient DCE for Tru64 UNIX. Because the **phnbk** source code is portable, you can build and run the **phnbk** server on a Tru64 UNIX system that has Gradient DCE for Tru64 UNIX installed, as well as on other DCE machines.

The sample **phnbk** client/server program demonstrates several aspects of cross-environment applications:

- Basic connectivity between a Microsoft RPC client and a DCE server.
- Client use of automatic binding in which the client gets the server binding information from the DCE Cell Directory Service. Alternatively, users enter a server's binding information as part of the command to start the client. Use the manual method to bypass the DCE Cell Directory Service or to select a specific server for use when several are available.
- The use of a portability file (**dosport.h**) that resolves differences between Microsoft RPC and DCE RPC.
- The use of portable server and client code. The server code builds and executes on Tru64 UNIX, Windows NT, and OpenVMS systems. The client code builds and executes on personal computers running the MS-DOS operating system, the Microsoft Windows NT operating system, and on Tru64 UNIX and OpenVMS systems.

13.7.1 Source Files for the phnbk Example

To run the example programs, copy the example source files to the Microsoft RPC and DCE platforms. The following list identifies the source files you need to build an executable client and server program for Gradient DCE for Tru64 UNIX.

- README file - describes how to build and run the example program
- **phnbk.idl** - Interface definition file
- **phnbk.acf** - Attribute configuration file
- **client.c** - Client program
- **server.c** - Server initialization code
- **manager.c** - Remote procedures
- **phnbk.txt** - **phnbk** database
- **phnbk.unix** - Makefile for UNIX client and server
- **dosport.h** - Microsoft RPC client portability file
- **phnbk.dos** - Makefile for MS-DOS client
- **phnbk.nt** - Makefile for Windows NT client
- **phnbk.com** - Command file to build client and server on OpenVMS systems\

13.7.2 Building the Tru64 UNIX phnbk Client and Server Programs

To build the **phnbk** client and server programs on a Tru64 UNIX system that has Gradient DCE for Tru64 UNIX installed, use **make** to build the executable client and server programs:

```
% make -f phnbk.unix ALPHA=_std1 alpha
```

This command creates a server program called **phnbkd** and a client program called **phnbk**.

13.7.3 Starting and Stopping the phnbk Server

To start the server **phnbkd**, enter the following command:

```
% phnbkd&
```

The server displays the binding information for each protocol sequence it is using. Three elements make up a server's binding information: a protocol sequence, a network address, and a transport endpoint. For example, a server might report the following binding information:

```
% phnbkd&
[1] 23789
ncacn_ip_udp:16.20.16.134.[1229]
ncacn_ip_tcp:16.20.15.134.[1474]
```

When you are done using the server program, stop it using the **kill** command.

13.7.4 Starting and Stopping the phnbk Client Program

To start the **phnbk** client (**phnbk**), use one of the following binding methods:

- Automatic binding. The client gets server binding information from the DCE Directory Service. The server must be running in the same DCE cell as the client. The following command starts the client using automatic binding.

  ```
  % phnbk
  Resolving binding through name server
  Server returned from name server is: ncacn_ip_tcp:16.20.16.134[]
  Valid commands are:

  (b)rowse - List next entry
  (r)eset - Reset to beginning of file
  (f)ind <string> - Find a substring
  (f)ind - Find next occurrence of <string>
  (q)uit - Exit program
  ```

- Manual binding. If the directory service is not available or you want to use a specific server, you can include the server's binding information as part of the command to start the client. You can use a complete or a partial binding. A partial binding consists of a protocol sequence and a network address, but does not include the server endpoint. If you do not include the endpoint, the client obtains it from the server host's endpoint map.

  The following command starts the client and uses a *complete* binding.

```
% phnbk ncan_ip_tcp:16.20.16.134\[1474\]
Valid commands are:

(b)rowse - List next entry
(r)eset - Reset to beginning of file
(f)ind <string> - Find a substring
(f)ind - Find next occurrence of <string>
(q)uit - Exit program
```

The following command starts the client and uses a *partial* binding.

```
% phnbk ncacn_ip_tcp:16.20.16.134
Valid commands are:

(b)rowse - List next entry
(r)eset - Reset to beginning of file
(f)ind <string> - Find a substring
(f)ind - Find next occurrence of <string>
(q)uit - Exit program
```

The phnbk client displays a menu of available commands that you enter to interact with the server. To stop the client, use the (q)uit command from the client menu.

13.8 The Echo Example Program

The Echo example program* (**echo**) demonstrates how a distributed application can secure itself using the GSSAPI security interface.

The **echo** example consists of a server program (**echo_server**) and a client program (**echo_client**). When **echo_server** is running, it waits for **echo_client** to attempt to connect over TCP/IP. Once a connection is established, user input from **echo_client** is transmitted across the network to the server and echoed back to the client.

When a user enters the **-s** switch, **echo_client** uses GSSAPI to authenticate itself to the server and to protect messages that flow from client to server. Messages in the reverse direction are not protected.

The **echo** example demonstrates how a distributed application:

- Creates GSSAPI server credentials with the **gss_acquire_cred( )** call
- Authenticates itself and creates a security context with the **gss_init_sec_context( )** and **gss_accept_sec_context( )** calls
- Protects individual messages cryptographically and verifies them using the **gss_seal( )** and **gss_unseal( )** calls

To build the **echo** example, copy the files in **/etc/examples/dce/gssapi** into a directory, edit **Makefile.echo** to match your environment (if necessary), and issue the following command:

```
% make -f Makefile.echo
```

You need to establish the echo server and client as DCE principals having principal names, accounts, and group and organization membership.

```
% dcecp
dcecp> principal create {echo_server echo_client}
dcecp> group add none -member echo_server
```

```
dcecp> group add none -member echo_client
dcecp> organization add none -member echo_server
dcecp> organization add none -member echo_client
dcecp> account create echo_server -group none -organization none \
> -password qwerty -mypwd -dce-
dcecp> account create echo_client -group none -organization none \
> -password xyzzy -mypwd -dce-
```

You must also create a keytable for managing the server authentication keys.

```
% rgy_edit

Current site is: registry server at /.../snafu_cell/subsys/dce/sec/master

rgy_edit=> ktadd -p echo_server -pw qwerty -f /tmp/echo_keyfile
rgy_edit=> quit
bye.

%
```

You can run the example from a single system or move **echo_client** and **echo_server** to two different systems. On the server, start **echo_server** using the following command syntax:

```
% echo_server [-p port] [-s server_name] [-f keytable_file]
```

The command arguments for the server are described in the next table.

Table 13-4: Server Options for the echo_server Command

Option	Description
-p port	Specifies the name or number of the TCP port on which the server listens for connection requests from clients. If you omit the -p switch, port 6000 is used.
-s server-name	Specifies a DCE principal name that the server uses to accept incoming connection requests that use GSSAPI authentication. The server needs access to a key corresponding to this principal name.
-f keytable-file	Specifies the pathname for the key table containing the principal's key. If you omit the -f switch, the DCE default key table is used. (You must run the server as root to use the default DCE key table).

You can try using another port if the server fails to start and produces an error like:

```
server: Can't bind local address
```

You can perform authenticated or unauthenticated client operations. To perform authenticated client operations, you must acquire DCE credentials with integrated login on an SIA-enabled system or by running the dce_login program. To perform unauthenticated operations, do not use the -s option to the echo_client command.

On the client, start echo_client using the following command syntax:

```
% echo_client [-h host] [-p port] [-s server-name]
```

The command arguments for the server are described in the next table.

Table 13-5: Client Options for the echo_server Command

Option	Description
-h host	Specifies the host name or IP address of the server machine. If you omit the **-h** switch, the client attempts to contact a server on the local system.
-p port	Specifies the name or number of the TCP port on which the server is listening. Specify the same port you specified to the server. If you omit the **-p** switch, the client attempts to contact a server on port 6000.
-s server-name	Specifies the DCE principal name of the server. Specify the same principal name you used when you started the server. If you omit the **-s** switch, GSSAPI is not used and the application operates as a simple, unsecured echo program. Specifying **-s** causes the client to authenticate itself to the server and to attach a cryptographically protected checksum to each message the client sends. The server validates the checksum before echoing the message.

Once the connection is open, each line you type to **echo_client** is sent across the network to the server and echoed back to the client. Press **<Ctrl/D>** to stop the client.

13.9 Time Provider Example Programs

The directory **/usr/examples/dce/dts** contains many example programs for various types of external time providers. These examples contain extensive information about how to build and use them. For additional information about the time provider interface, see the *OSF DCE Application Development Guide*.

13.10 The Serviceability API Sample Program

The **hello_svc** program provides a simple demonstration of the DCE Serviceability API. When executed, it writes a “Hello world” message to standard error via the serviceability interface.

The program was developed during the writing of the *OSF DCE Application Development Guide* chapter on serviceability, and is included with the DCE software as a very simple demonstration of the interface.

13.10.1 Building the Program

To build the example, copy the files from **/usr/examples/dce/svc** into a writeable directory and issue this command:

```
% make _f Makefile.hello_svc
```

Once you have built hello_svc, execute it by typing this command with no arguments:

```
% hello_svc
```

You should see two messages similar to these:

```
1994-06-10-13:07:33.628+00:00I----- ./hello_svc NOTICE hel main 0xa448c444
Hello world
1994-06-10-13:07:33.628+00:00I----- ./hello_svc NOTICE hel main 0xa448c444
Hello world
```

The message is printed twice because it is routed to standard error twice: once via a call to **dce_svc_routing( )** within the program, and again by the "attributes" field in the message definition in the hel.sams file.

For more information on Serviceability functionality, the *OSF DCE Application Development Guide - Core Components.* See also the **log.8dce** reference page (about the **dcecp log** object, through which the DCE components serviceability routes and settings are managed) in the *OSF DCE Command Reference*.

13.11 The Generic Sample Application

The generic sample DCE client/server application includes extensive examples of ACL management, serviceability code, security setup, and signal handling. It also has the necessary initialization and cleanup code. The manager code (**sample_manager.c)** consists of one generic remote call that does no actual work, but which does make use of the ACL manager and the serviceability code.

13.11.1 Building the Sample Application

To build the sample application program, copy the source files from **/usr/examples/dce/generic_app/***. Use the following command to build the application:

```
make _f Makefile.generic_app
```

13.11.2 Installing the Sample Application

Before you can run the sample application, you must install **sample_client** and **sample_server** on the machines you want to use. This installation involves these steps:

1. Adding the client and server principals and server group to the Security registry.
2. Creating a keyfile to be used by the server.
3. Creating a CDS namespace entry for the server to export its binding information to (and for the clients to import binding information from).
4. Setting up the correct permissions on the namespace entry to allow the server to use it (that is, to write to it) correctly.

Assuming that the server's principal name is sample_server and that the client's principal name is sample_client, you should perform these steps as follows:

1. Log in as the cell administrator:

```
$ dce_login cell_admin -dce-
```

You must first login as the cell administrator to be able to execute the registry operations in step 2.

NOTE: The password at your site is probably different from that given above (as the last parameter). For further information about the use of **dce_login**, see the *OSF DCE Administration Guide.*

---

2 Add the server and client principals to the registry, and set up the server's keyfile:

```
% dcecp
dcecp> group create sample_servers
dcecp> user create sample_server -g sample_servers -org none
> -pass server_password -mypwd -dce-
dcecp> user create sample_client -g none -org none
> -pass client_password -mypwd -dce-
dcecp> keytab create /.:/hosts/mccann/config/keytab/sample_keytab
> -storage /tmp/sample_keytab
> -data {sample_server plain 1 server_password}
> -noprivacy -local
dcecp>
```

NOTE: *server_password* and *client_password* are the passwords that you assign to the server and client, respectively. You can substitute any other values but be sure to remember these values: you need to use them to perform a **dce_login** operation before executing the client and server programs. For further information about **dcecp**, see the *OSF DCE Administration Guide - Core Components* and the *OSF DCE Command Reference*.

The name of the server's keyfile, /tmp/sample_keytab, is specified by the value of the KEYTAB constant in the sample_server.c file; the name you give to the keytab subcommand must be identical to the value of this constant.

3 Create the CDS entry to be used to hold the server's binding information. For example:

```
% dcecp -c create directory /.:/sample
% dcecp -c rpcentry create /.:/sample/sample_server_entry
```

You can substitute any legal CDS name for sample.

4 Set up the ACL on the entry to allow access to the server:

```
% dcecp
dcecp> acl modify /.:/sample/sample_server_entry -entry \
> add {user sample_server rwdtc}
dcecp> exit
```

NOTE: **sample_server** is the principal name used in the previous steps and must be identical to the value of the *principal_name* argument you specify on the command line to **sample_server**.

You have now installed the sample application.

13.11.3 Running the Sample Application

This section describes how to run the server and client.

Before you run the server you must create in the local directory, a subdirectory called **db_sample_acl**. This directory is where the sample application's backing store database files will be created. The pathname to these files is determined by the value of the ACL_DB_PATH constant at the top of the **sample_server.c** file; you can change this value if you want to.

Invoke the server as follows:

sample_server *principal_name CDS_dir_name/*

NOTE: There is a /(slash) after the directory name.

where:

principal_name	The server's principal name. An account must be in the registry for this principal for the program to run successfully. Note that this name is not specified in the program source; it is determined solely by the user, who must make sure that the name he or she specifies here is the same as the one set up in the registry.
CDS_dir_name	The full name (terminated by a / (slash)) of the CDS directory in which the server's namespace entry is to be located; the bindings are exported to this directory. Note that this argument is NOT the name of the server entry which is determined by the value of the constant DEFNAME, defined in sample_server.c: the server entry is created in the *CDS_dir_name directory.*

For example (with setup done as described in first section):

```
./sample_server sample_server /.:/sample/
```

At present, the server's serviceability messages are routed by default values coded at the top of the **sample_server.c** file. The default behavior sets full debugging and routes everything to **stderr**. If you compile the server as is, you see lots of messages appearing on your screen when you run it (For an example, see the end of the README file.) To change this behavior, you must change the hard-coded defaults, because currently there is no way to change routing via the command line.

13.11.3.1 Running the Client

Before running the client you must first set the environment variable RPC_DEFAULT_ENTRY to the value of the full name of the server's CDS name entry. For example (with setup done as described in first section):

```
setenv RPC_DEFAULT_ENTRY /.:/sample/sample_server_entry
```

You must be logged in via **dce_login** as the **sample_client** principal to properly allow the client to do what it needs to do. This is because the only principal who is given any meaningful permissions on the objects managed by the application is the owner who is defined at the top of **server_sample.c** to be **sample_client**.

The client is invoked as follows:

sample_client *object_name* **| kill**

object_name	The name of the object you want the client to bind to. Note that this is not the entry name of some exported entity; it is some object managed by the server and held in a backing store. Specify the simple object name, the client will try to bind through the RPC_DEFAULT_ENTRY value.
kill	A keyword that specifies the server be killed via a call through the remote management interface.

You can try any of three command forms (because at present there are only two objects set up by the server).

To bind via the junction to the mgmt object and view its contents, enter:

```
./sample_client sample_object
```

To bind to the sample object and view its contents, enter:

```
./sample_client server_mgmt
```

To kill the server via the remote management interface, enter:

```
./sample_client kill
```

13.11.4 What the Sample Application Does

You can run the client in either of two modes: you can specify that the server be killed or you can specify a single object to bind to. The object name is specified by a namespace pathname, but neither of the two possible objects is a namespace entry. Instead, the sample application implements a "junction" located at its server entry in the namespace, and clients bind to objects through this junction.

When the client tries to bind to the overqualified CDS entry formed by concatenating the specified object name to the server entry name it obtains a partial binding to the server. The client then makes a call to the remote bind operation with that binding, ostensibly to get the object UUID of the object whose name was specified (to bind to) when the client was invoked. These objects reside in a backing store database. The remote call makes its way by the familiar procedure to the server; the application's **name_to_object( )**

routine (defined in **sample_bind.c**) then simply looks up the desired object UUID by accessing the name-indexed backing store. When the remote call completes, the client has a full binding and the desired object UUID.

13.11.5 Viewing the Server ACL

With the sample_server running, you can also access the server's ACL managers using **dcecp. For example, to get a list of the contents of the ACL, enter:**

```
dcecp -c acl show /.:/sample/sample_server_entry/sample_object
```

This command produces the following output:

```
{user sample_client rwdctx}
```

The same commands can be used to bind to and list the contents of the **server_mgmt** ACL.

The README file contains sample output from the Generic Application.

For further information the acl object in **dcecp**, see the *OSF DCE Command Reference*.

13.11.6 Notes

A detailed explanation of the operation of the ACL management code is in the *OSF DCE Application Development Guide - Introduction and Style Guide.*

The sample application does not use the OSF DCE **dced** facilities, by which a DCE application can be registered (either via calls to the **dced_server_** routines or via **dcecp** by a system administrator) with **dced**, and then, by means of calls to the **dce_server_** routines, to have **dced** do almost all of its namespace and security initialization for it. For more information on the **dced_server_** and **dce_server_** routines and their use, see the *OSF DCE Application Development Guide - Introduction and Style Guide* and the *OSF DCE Application Development Guide - Core Components.*

13.12 Object Oriented idl Programs

This section describes how to build and run four example programs that demonstrate the use of C++ idl extensions. The four programs are

- The **account** example program
- The **accountc** example program
- The **card** example program
- The **stack** example program

13.12.1 Preparing to Run the Example Programs

The C++ example programs require C++ software to be installed and configured on the client and server machines.

Establish your environment for building and running the example programs as follows:

1 Copy the four example programs into a directory tree with a root name of **./idlcxx.**

```
% cp _R /usr/examples/dce/rpc/idlcxx/* .
```

The **./idlcxx** directory has the following files and subdirectories

README	A file containing instructions relevant to all example programs
account	A directory with the **account** example program sources
accountc	A directory with the **accountc** example program sources
card	A directory with the **card** example program sources
idlcxx_setup	A shell script that creates a CDS directory and sets some ACLs
stack	A directory with the **stack** example program sources

2 Log in to the DCE cell as **cell_admin** and run the **idlcxx_setup** shell script. This creates a test directory in the Cell Directory Service and establishes necessary ACL entries.

```
% dce_login
Enter Principal Name: cell_admin
Enter Password:
% idlcxx_setup
%
```

Once a server has been built and is executing, you can start and stop client programs as many times as desired. You can remove the executable client and server programs from a directory using the command:

```
% make clean
```

13.12.2 The account Example Program

The **account** example program tests inheritance, binding to an object using another interface, binding to an object with an unsupported interface, and the reflexive, symmetrixc, and transitive relation properties of the **bind( )** API. A Savings interface is derived from an Account interface. A now/Account implementation class is derived from the Savings and Checking interfaces. A oldAccount implementation class is derived from the Savings but not the checking class which implies that an oldAccount does not support a Checking interface.

This example program requires C++ software to be installed and configured on the client and server machines.

Build this example program by entering the command:

```
% make
```

Start the server by entering the command:

```
% ./server &
```

Once the server is running, you can run the client on the same host, or on any other host in the network that is configured to run in the same cell as the server host. Before running the client, you must define an environment variable on the client system that can be used to locate the server binding information in the namespace during the auto-handle process:

```
% setenv RPC_DEFAULT_ENTRY /.:/subsys/DEC/examples/account_server
After you define the environment variable, run the client with the command:
% client
```

The client binds to an object, uses different interfaces, and binds to dynamic interfaces. It also exercises bind relation properties.

13.12.3 The accountc Example Program

The **accountc** example program tests the same properties as the **account** program (see *Section 13.12.1 on page 180*), but uses the C interfaces for all the APIs.

This example program requires C++ software to be installed and configured on the client and server machines.

Build this example program by entering the command:

```
% make
```

Start the server by entering the command:

```
% ./server &
```

Once the server is running, you can run the client on the same host, or on any other host in the network that is configured to run in the same cell as the server host. Before running the client, you must define an environment variable on the client system that can be used to locate the server binding information in the namespace during the auto-handle process:

```
% setenv RPC_DEFAULT_ENTRY /.:/subsys/DEC/examples/accountc_server
```

After you define the environment variable, run the client with the command:

```
% client
```

13.12.4 The card Example Program

The **card** example program tests the passing of C++ objects as parameters using the **[cxx_delegate]** attribute and the polymorphism property of the base class. A Player implementation class is a generic sports card class. Derived from Player are a BaseballPlayer class and a BasketballPlayer class. The application interfaces with the Player class to invoke virtual operations in the derived class.

This example program requires C++ software to be installed and configured on the client and server machines.

Build this example program by entering the command:

```
% make
```

Start the server by entering the command:

```
% ./server &
```

Once the server is running, you can run the client on the same host, or on any other host in the network that is configured to run in the same cell as the server host. Before running the client, you must define an environment variable on the client system that can be used to locate the server binding information in the namespace during the auto-handle process:

```
% setenv RPC_DEFAULT_ENTRY /.:/subsys/DEC/examples/card_server
```

After you define the environment variable, run the client with the command:

```
% client
```

13.12.5 The stack Example Program

The **stack** example program tests the passing of C++ objects as parameters using the **[cxx_delegate]** attribute and a user defined Stack class. This test implements a reverse Polish notation algorithm where the binary arithmetic operations are performed on the server.

This example program requires C++ software to be installed and configured on the client and server machines.

Build this example program by entering the command:

```
% make
```

Start the server by entering the command:

```
% ./server &
```

Once the server is running, you can run the client on the same host, or on any other host in the network that is configured to run in the same cell as the server host. Before running the client, you must define an environment variable on the client system that can be used to locate the server binding information in the namespace during the auto-handle process:

```
% setenv RPC_DEFAULT_ENTRY /.:/subsys/DEC/examples/stack_server
```

After you define the environment variable, run the client with the command:

```
% client
```

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

X