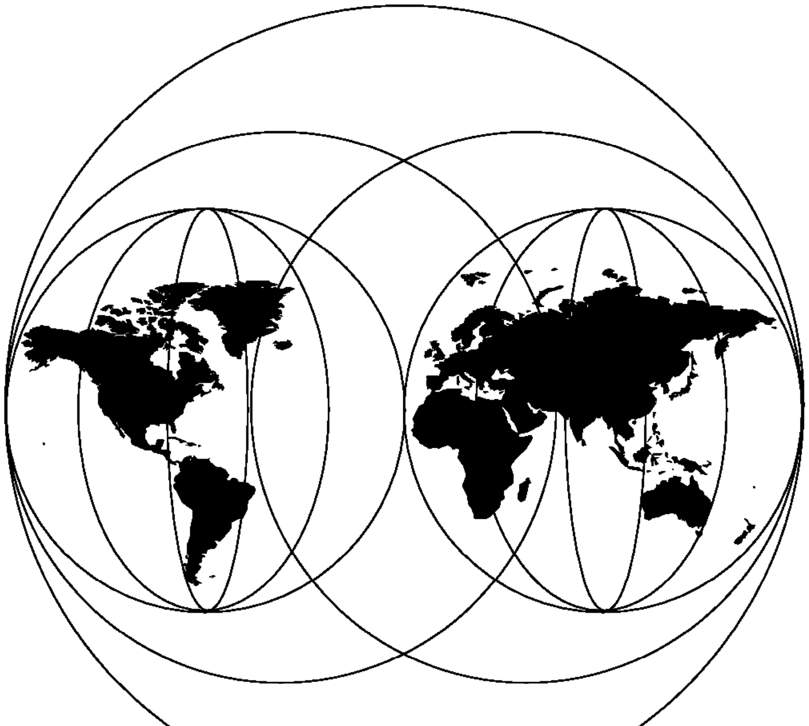
International Technical Support Organization SG24-4616-00

Understanding OSF DCE 1.1 for AIX and OS/2

October 1995



IBM

International Technical Support Organization
Austin Center

International Technical Support Organization

SG24-4616-00

Understanding OSF DCE 1.1 for AIX and OS/2

October 1995

Take Note!

Before using this information and the product it supports, be sure to read the general information under "Special Notices" on page xv.

First Edition (October 1995)

This edition applies to the IBM DCE Version 2.1 Product Family for AIX Version 4.1 and the IBM DCE 2.1 for OS/2 WARP Beta Program.

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

An ITSO Technical Bulletin Evaluation Form for reader′s feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept. JN9B Building 821 Internal Zip 2834
11400 Burnet Road
Austin, Texas 78758-3493

Abstract

This publication is intended to help customers and system engineers understand the Open Software Foundation (OSF) Distributed Computing Environment (DCE) and its implementation on AIX and, in beta release, on OS/2 WARP. It explains, at a high level, the functions and features of OSF DCE 1.1 to the level of detail necessary to understand how everything works, including Tcl, threads and RPC programming. Differences in previous DCE versions are explicitly mentioned. It also provides step-by-step instructions on how to install and customize an intercell scenario involving AIX and OS/2 WARP systems.

This document is a complete rework and replacement of the document *OSF DCE for AIX, OS/2 and DOS Windows Overview* (order number GG24-4144).

(245 pages)

Contents

Figures

Tables

Special Notices

AIX	AIX/6000

Preface

The purpose of this document is to provide a comprehensive introduction to the OSF DCE at level 1.1. It explains, in high-level terms, all the components and features to the level of detail necessary to understand how everything works.

Three different types of users may be concerned with DCE:

- End-users
- Administrators
- Application developers

End-users need not know much of DCE. The applications they are using should hide the distributed nature of the environment. For them, the first chapter may be useful, if they want to know what DCE is.

Administrators need to understand many details and how *all* the components work. The community of DCE administrators will grow when other products, such as LAN Server for OS/2 or DSOM, become integrated with DCE. This redbook can be a good tutorial for novice users who need to learn the internals of DCE and for experienced administrators who need to understand the new functions.

Developers need to know all the details, including RPC and threads programming, if they are using the low-level DCE RPCs. They can then get the same benefits from this redbook as explained above for an administrator. However, it is more likely that developers of distributed applications will use a higher level programming tool that uses DCE beneath it and hides much of its complexities.

This redbook replaces the document *OSF DCE for AIX, OS/2 and DOS Windows Overview* (order number GG24-4144). Although we do not cover the DCE for DOS Windows, we kept some information on the currently available product, IBM DCE for DOS Windows (OSF DCE level 1.0.2), from the old redbook when we discuss platform-specific implementations of the DCE components.

How This Document is Organized

The first part, after the introduction, explains the DCE components with which a DCE administrator has to deal the most: the CDS, the Security Service, the DTS, and DFS. Then we describe, in a second part, how to install and configure a cell with AIX and OS/2 systems followed by a discussion of compatibility and migration issues. The third part covers Tcl, RPC and Threads, which are of importance only to the reader who is interested in creating dcecp scripts and/or in RPC program development. The chapters are the following:

- Chapter 1, "Introduction"

  This chapter discusses DCE and the role it is playing in IBM′s strategy for distributed computing, the Open Blueprint. It also explains the OSF and how it acquires new technology. Finally, it provides an overview of the DCE components, the new features of OSF DCE 1.1 and product information on the AIX and OS/2 platforms.

- Chapter 2, "Directory Service"

This chapter explains the concept of a DCE cell and the DCE global namespace and how the Directory Service is implemented.

- Chapter 3, “Security Service”

  This chapter explains all the components of the Security Service that are involved in authenticating and authorizing users and applications within a cell and across cell boundaries.

- Chapter 4, “Distributed Time Service”

  The DTS is an often neglected component. This chapter explains the necessity of a time service and explains how it needs to be correctly laid out in a cell.

- Chapter 5, “Distributed File Service”

  This chapter gives an overview over the DFS and explains its concepts without details that are covered in separate redbooks.

- Chapter 6, “Installation and Configuration of DCE”

  This chapter provides step-by-step instructions on how to install DCE and configure an intercell scenario involving AIX and OS/2 workstations.

- Chapter 7, “Migration and Compatibility”

  This chapter discusses compatibility between the different OSF DCE releases and provides help for migrating OSF DCE 1.0.x environments to OSF DCE 1.1 environments, particularly on the AIX platform.

- Chapter 9, “DCE Control Program and Tcl”

  This chapter explains the Tool Command Language used by the new DCE control program. It can be considered a Tcl programming and general dcecp usage tutorial.

- Chapter 10, “Remote Procedure Calls”

  This chapter explains how RPC applications work and how an RPC client finds its way to a particular RPC server function. It also discusses how Security Service features are exploited for RPC and shows the components and steps involved in RPC application development. It can be considered a high-level DCE application-development tutorial.

- Chapter 11, “Threads”

  This chapter introduces the concepts of threads and threads programming. It discusses mutexes and condition variables, scheduling, exception handling and specialities in a UNIX environment, such as signals.

- Appendix A, “DCE Application Examples”

  This appendix lists example application code that comes with the DCE product and explains what these samples do and where to find them. These examples are very useful for novice DCE RPC programmers.

Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this document.

General DCE Books

- *Networking Blueprint, Executive Overview*, GC31-7057

- *Understanding DCE Concepts*, GC09-1478
- *OSF DCE User′s Guide and Reference (Prentice Hall)*, SR28-4992
- *OSF DCE Administration Reference (Prentice Hall)*, SR28-4993
- *OSF DCE Application Development Guide (Prentice Hall)*, SR28-4994
- *OSF DCE Application Development Reference (Prentice Hall)*, SR28-4995
- *Understanding DCE (O′Reilly & Associates)*, SR28-4855

DCE Version 1.3 for AIX

- *DCE V1.3 for AIX Release Notes*, GC23-2434
- *DCE V1.3 for AIX User′s Guide and Reference*, SC23-2729
- *DCE V1.3 for AIX Administration Guide -- Core Services*, SC23-2730
- *DCE V1.3 for AIX Administration Guide -- Extended Services*, SC23-2731
- *DCE V1.3 for AIX Administration Reference*, SC23-2732
- *DCE V1.3 for AIX Application Development Guide*, SC23-2733
- *DCE V1.3 for AIX Application Development Reference*, SC23-2734
- *DCE NFS to DFS Authenticating Gateway V1.3 for AIX*, SC23-2735
- *NetView for DCE and Encina Manager Guide V1.3*, SC23-2736
- *AIX HACMP for DCE and Encina Guide V1.3*, SC23-2737
- *AIX DCE Getting Started V1.3*, SC23-2477
- *AIX DCE and OS/2 DCE Message Reference*, SC23-2583

DCE Version 2.1 for AIX

- *Introduction to DCE V2.1 for AIX*, SC23-2796
- *DCE V2.1 for AIX: Getting Started*, SC23-2797

These are the only printed manuals. The documentation basically comes with the program components in softcopy form only. These manuals can be printed from within the ASCII viewer in ASCII format or from within the IPF/X graphical softcopy browser in PostScript format. The following softcopy books are available:

- *Introduction to DCE*
- *DCE for AIX Getting Started*
- *DCE for AIX Administration Guide*
- *DCE for AIX Administration Command Reference*
- *DCE for AIX Application Development Guide - Introduction*
- *DCE for AIX Application Development Guide - Core Services*
- *DCE for AIX Application Development Guide - Directory Services*
- *DCE for AIX Application Development Reference*
- *DCE for AIX DFS Administration Guide and Reference*
- *DCE for AIX NFS/DFS Authenticating Gateway Guide and Reference*

DCE Version 1.2 for OS/2 and Windows

- *IBM DCE for OS/2: Guide to Planning, Installation and Configuration*, S96F-8502
- *IBM DCE for OS/2: Administrator's Guide*, S96F-8504
- *IBM DCE for OS/2: Administrator's Command Reference*, S96F-8505
- *IBM DCE for OS/2: Application Developer's Guide*, S96F-8506
- *IBM DCE for OS/2: Application Developer's Reference*, S96F-8507
- *IBM DCE for OS/2: Master Index*, S96F-8615
- *IBM DCE Client for Windows User's Guide*, S96F-8622
- *IBM DCE SDK for Windows Guide and Reference*, S96F-8623

International Technical Support Organization Publications

- *IBM DCE Cross-Platform Guide*, GG24-2543
- *Using and Administering AIX DCE 1.3*, GG24-4348
- *Developing DCE Applications for AIX, OS/2 and Windows*, GG24-4090
- *The Distributed File System (DFS) for AIX/6000*, GG24-4255
- *Elements of Security: AIX 4.1*, GG24-4433
- *Using Network Security Program on AIX, OS/2, and DOS Platforms*, GG24-4149

A complete list of International Technical Support Organization publications, known as redbooks, with a brief description of each, may be found in:

International Technical Support Organization Bibliography of Redbooks, GG24-3070.

To get a catalog of ITSO redbooks, VNET users may type:

```
TOOLS SENDTO WTSCPOK TOOLS REDBOOKS GET REDBOOKS CATALOG
```

A listing of all redbooks, sorted by category, may also be found on MKTTOOLS as ITSOCAT TXT. This package is updated monthly.

How to Order ITSO Redbooks

IBM employees in the USA may order ITSO books and CD-ROMs using PUBORDER. Customers in the USA may order by calling 1-800-879-2755 or by faxing 1-800-284-4721. Visa and MasterCard are accepted. Outside the USA, customers should contact their local IBM office. For guidance on ordering, send a PROFS note to BOOKSHOP at DKIBMVM1 or E-mail to bookshop@dk.ibm.com.

Customers may order hardcopy ITSO books individually or in customized sets, called BOFs, which relate to specific functions of interest. IBM employees and customers may also order ITSO books in online format on CD-ROM collections, which contain redbooks on a variety of products.

ITSO Redbooks on the World Wide Web (WWW)

Internet users may find information about redbooks on the ITSO World Wide Web home page. To access the ITSO Web pages, point your Web browser to the following URL:

```
http://www.redbooks.ibm.com/redbooks
```

IBM employees may access LIST3820s of redbooks as well. The internal Redbooks home page may be found at the following URL:

```
http://w3.itsc.pok.ibm.com/redbooks/redbooks.html
```

Acknowledgments

This project was designed and managed by:

Rolf Lendenmann	IBM ITSO Austin

The authors of this document are:

Rolf Lendenmann	IBM ITSO Austin
Joe Vicini	IBM USA
Gerardo Vega	IBM Mexico
Urs Weber	IBM Switzerland

This document is a complete rework and replacement of the document *OSF DCE for AIX, OS/2 and DOS Windows Overview* (GG24-4144). This previous document was created by:

Edgar Villegas (Advisor)	IBM ITSO Austin
Mathilde Ten Seldam	IBM France
Ricardo Haragutchi	IBM ITSO Raleigh
Salvatore La Pietra	IBM Germany

This publication is the result of a residency conducted at the International Technical Support Organization, Austin Center.

We would like to offer our thanks to **Marcus Brewer** of the ITSO Austin and to the IBM-Austin DCE development team for their invaluable advice and assistance in the reviewing of this document.

Chapter 1. Introduction

This document is about the IBM implementation of the *Open Software Foundation Distributed Computing Environment (OSF DCE) Version 1.1*. The purpose of this document is to provide the reader detailed, high-level information on DCE concepts and the results of practical experience with the IBM DCE Version 2.1 for AIX (product) and OS/2 (beta).

1.1 OSF Distributed Computing Environment

The OSF Distributed Computing Environment (DCE) is a set of integrated services designed to support the development and use of distributed applications. OSF DCE is operating-system independent and provides a solution to the problem of sharing resources in a heterogeneous, networked environment. This is accomplished by providing the services necessary to create an environment where a group of networked machines can share and manage resources. This allows for efficient use of present technology and for new technology to be incorporated as it becomes available.

DCE provides interoperability and portability across diverse operating systems and hardware platforms. DCE is a complete framework on which you can develop and maintain your distributed applications and services in an environment that you can scale to address the changing requirements of your enterprise.

1.2 IBM′s Open Blueprint

Open Blueprint is IBM′s functional description of open distributed computing. The Open Blueprint does not specify software packages but rather, the functional modules, software principles/guidelines and certain important interfaces. The goal of the Open Blueprint is to provide a single system view of the network.

Existing products provide many of the functions described in the Open Blueprint. Additional functionality will be provided through the introduction of new products. IBM has encouraged the integration of multivendor products into the Open Blueprint by publishing the standards for each component. IBM worked closely with international standards organizations, industry consortia, customers, and industry leaders to determine the most widely accepted standards available today. Products that are written to the Open Blueprint provide designated interfaces and protocols. Products that use resource managers defined within the Open Blueprint could be integrated. For example, for applications and resource managers that use security services from the distribution services module of the Open Blueprint, a single user password would be possible. This kind of integration helps to make the distributed environment more transparent for both the user and the application developer.

The four roles of the Open Blueprint are:

1. To help customers develop their architectures and organize products in an open distributed environment.
2. To describe IBM′s direction for open distributed computing.
3. To guide developers in product design and implementation.

4. To provide a context for incorporating new technology into a distributed environment.

OSF DCE plays an important role in the Open Blueprint. For example, the Distributed Services module consists mainly of the OSF DCE core services, and the Data Access Services standard for the byte-stream API is the DCE Distributed File Service. Figure 1 presents the Open Blueprint Model.

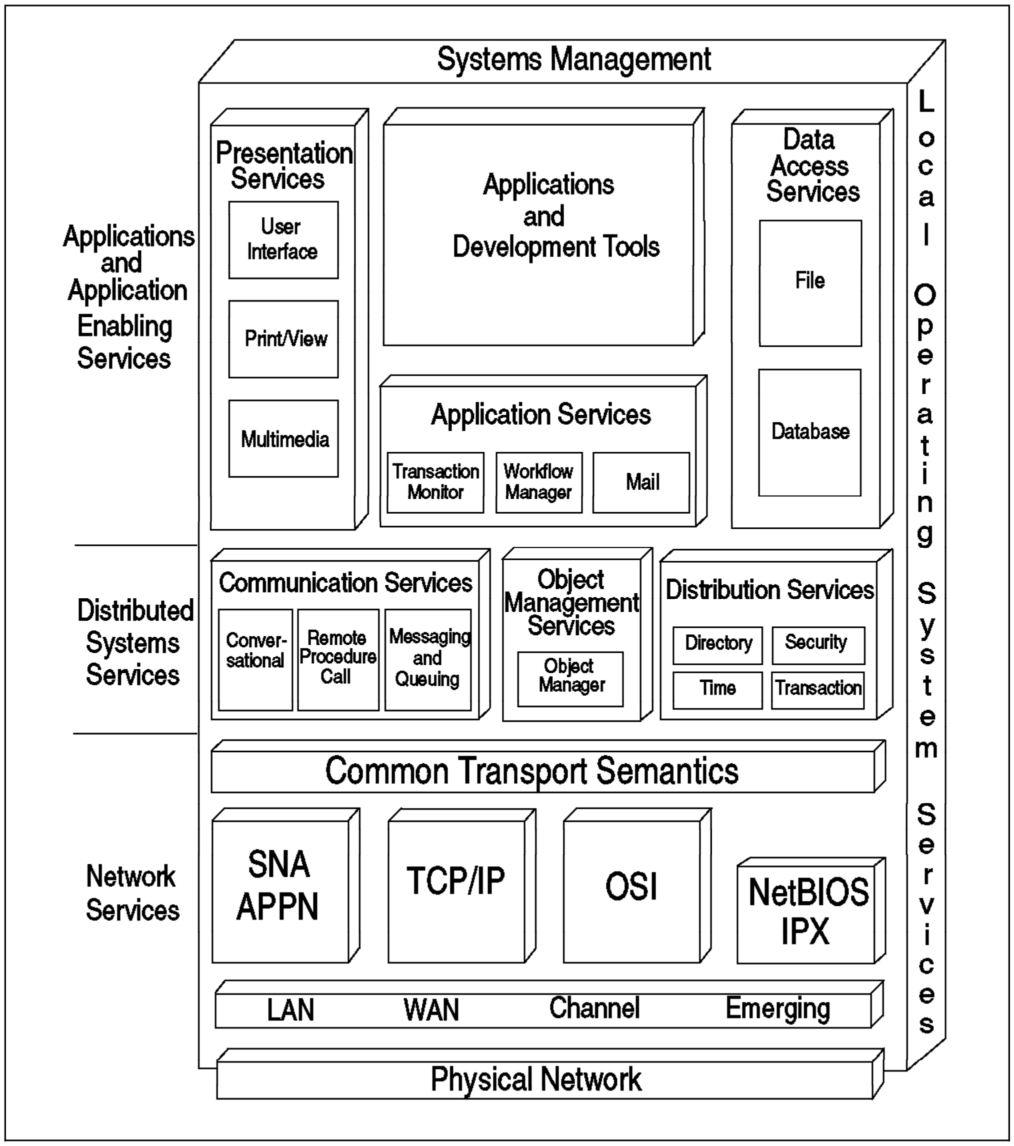


Figure 1. IBM Open Blueprint Functional Description

The Open Blueprint specifies IBM′s strategy for open distributed computing. Being part of that, DCE will be IBM′s strategic technology for communication and distribution services. This means that even existing and well-known applications or tools will get DCE support behind the scenes. An end-user will usually not be affected, but administrators of these products will need to have DCE administration skills.

DCE will provide the following benefits for customers moving to DCE-enabled products:

- Global resource access and administration

- Enterprise-wide integrated security
- One user ID, one password per user for all services
- Single administration model
- Better interoperability between products and platforms

Although not formally announced but mentioned by IBM speakers at trade shows and press conferences, the following products will be the first to incorporate DCE technologies:

- DB2
- MQSeries
- DSOM
- LAN Server
- Distributed Printing (Palladium)
- NetSP
- RACF

More products will follow. Some applications were built as distributed applications with DCE technology. IBM′s distributed Online Transaction Processing (OLTP) products, ***Encina*** and ***CICS for AIX***, are built on top of DCE. Many other DCE applications are available on IBM platforms from vendors. See 1.7, "IBM DCE Product Information" on page 14 on how to get the catalog of DCE applications (and development tools) that the OSF is publishing with quarterly updates.

1.3 The Open Software Foundation

The Open Software Foundation (OSF) was incorporated in 1988 as a non-profit organization to support the research, development and delivery of vendor-neutral technology and industry standards. The OSF is a member-supported organization open to any group that provides or uses open systems. OSF sponsors include International Business Machines Corporation, Bull Worldwide Information Systems, Hitachi Corporation, Hewlett-Packard Corporation, Digital Equipment Corporation, and Sun-Soft Corporation.

The OSF is a host for industry-wide open systems research and development. Users, software developers and hardware vendors share resources and ideas with the goal of improving the scalability, probability and interoperability of computer systems.

1.3.1 Open Software Mall

The Open Software Mall is a world-wide web software repository being implemented by OSF to facilitate the distribution of innovative open systems technology. The Mall will expedite the distribution of all of OSF′s software and services, including technology from the Research Institute, from Advanced Technology Offerings (ATO), Pre-Structured Technologies (PST) and Request for Technologies (RFT) as well as relevant technology from OSF members and from the research community. The Mall will also be a convenient vehicle to make documentation, plans, specifications, and test suites available. The Mall is accessible under the following URL:

```
http://www.osf.org/mall
```

At this time, the Mall is still under construction. At its inception, the Open Software Mall will include four plazas: an OSF Research Institute plaza, an

Advanced Technology Offering (ATO) plaza, a Pre-Structured Technology (PST) plaza, and an OSF Professional Services plaza. It is intended that additional plazas will be added in the future.

1.3.2 OSF Technology Development Process

To develop OSF open systems technology, the OSF uses four methods which we describe in the following sections.

1.3.2.1 Advanced Technology Offering (ATO)

An ATO develops technology to a sufficient degree of maturity that it can be effectively explored by end users, ISV′s and vendors. The ATOs are freely available from the Mall for non-commercial use. The target audience is the research and academic communities from whom investors hope to obtain early and low-cost feedback. An ATO technology that proves appealing can be the starting point for an OSF Pre-Structured Technology (PST) or for individual product development activities in the sponsor companies.

ATOs require a minimum of three investors contributing $100,000 for up to six engineers. Investors get unrestricted commercial rights. At this time, the following ATOs are underway:

- ***DCE Web ATO*** — Enables web servers and clients to employ OSF′s DCE for improving naming, security and access control. Investors are AT&T, HP and Hitachi.
- ***Microkernel Specification and Test Suite ATO*** — Will assure that microkernel implementations support a common API. Conformance and performance test suites will be included as part of this ATO. Investors are IBM, Hewlett-Packard, Hitachi, and DEC.
- ***JAVA Evaluation and Porting ATO*** — JAVA technology from Sun Microsystems enables Web distribution of smart data, new protocols, animations, and other active information in a secure, architecture-portable fashion. Investors are Sun, HP, AT&T, and Novell.

ATOs are expected to jumpstart collaboration around technology from industry, academia or the OSF Research Institute, providing a counterpoint to the commercial PSTs.

1.3.2.2 Pre-Structured Technology (PST)

The PST process is a new process that provides a method for working on existing technology and bringing it to market.

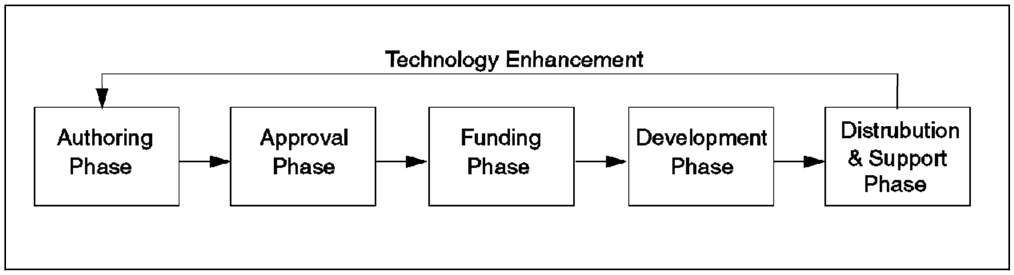


Figure 2. Open Software Foundation PST Process

As outlined in Figure 2, the OSF PST process has five phases:

1. Authoring Phase — Companies work together to form a proposal for submission to the OSF. This phase is open to both OSF members and non-members alike and may be as public or private as the authors prefer.
2. Approval Phase — Begins when a proposal is submitted to the OSF. OSF checks the proposal for fit with OSF′s open systems technology map, completeness, viability, and pro-competitiveness. The goal is to prepare the proposal for submission and evaluation by OFS′s board.
3. Funding Phase — Begins after OSF′s board approves a PST proposal. The opportunity to participate as a project sponsor is opened to all OSF sponsor companies.
4. Development Phase — Actual work on the PST is undertaken by the Prime Contractor. Work is directed by representatives of the project sponsor companies, the end-user community and the project steering committee. OSF acts as the overall program and project manager.
5. Distribution and Support Phase — The completed offering is licensed. Plans for any product support efforts are also completed and implemented.

1.3.2.3 Request For Technology (RFT)

The RFT process is used to search for new technology to provide open systems solutions. The difference between the PST and RFT process is the technology selection process. In RFTs, the technology is selected through an open call to the industry, and then the OSF staff and consultants choose the best technology from those received. In the PST process, the sponsors of a project bring the technology to the OSF and then work with the OSF to collaborate on common technology areas for development-cost savings. The PST process allows more projects to be under development at one time because the work load is shared by the prime contractors. OSF′s sponsors see more value in cost sharing and savings than in a centralized location for source code creation and delivery.

1.3.2.4 The OSF Research Institute (RI)

The OSF RI provides a path for technology from academia, industry and government agencies to be used in future open system solutions. The OSF RI investigates the fundamental software technology needed to improve open systems. The RI conducts collaborative research with academia and industry to create complete prototypes of technologies that can be used as a basis for further research. The output of the RI process may become the basis for a PST.

1.3.3 Special Interest Groups (SIG) and Workgroups

A Special Interest Group represents a group of OSF members focusing on a particular technology issue. The purpose of the DCE SIG is to provide input and assistance to the OSF in the area of distributed computing. Specifically, the SIG is chartered:

- To propose enhancements to OSF to the existing DCE offering and to help prioritize their relative importance.
- To make recommendations as to which new or emerging technologies should be included in some future version of DCE.
- To provide feedback to OSF on the business issues related to pricing, packaging, licensing, and marketing of the DCE technologies.
- To provide a liaison between OSF and various standards forums.
- To insure that technical issues which relate to both the DCE SIG and other OSF SIGs are resolved quickly and appropriately and in a way which does not compromise the success of the DCE.

- To provide a forum in which issues and status related to the progress and status of the DCE development can be communicated to all SIG members.

There are no requirements for attendance at the SIG meetings, except OSF membership.

The SIG has a number of working groups, each of which is focused on a particular area. In order to create a new working group, you must make a formal proposal (just discuss your idea) at a SIG meeting, and if there are enough people who want to participate, a new working group can be established.

A list of the current working groups, their chairperson, and the name of the distribution list for each one can be obtained by sending a note to `direct@osf.org`. You can also request to be added to one of the distribution lists.

1.3.4 Application Environment Specification (AES)

An AES is the detailed specification for an OSF technology component. The purpose of the AES/Distributed Computing (AES/DC) is to provide a portability guide for DCE applications and a conformance specification for DCE implementations. The different AES volumes specify the DCE model, services, interfaces, and protocols.

1.3.5 Request For Comment (RFC)

The DCE RFC series of technical notes is intended as an *online forum* for the OSF′s DCE Special Interest Group (DCE SIG) to share information about the DCE and related technologies. It is inspired by, and in many ways intended to resemble, its highly successful precursor, the Internet RFC series.

Internet RFCs have a highly-developed relationship with the Internet standardization process. Experimental protocols and proposed standards are published as (possibly draft) Internet RFCs and are subject to commentary before they progress along the standardization track. All Internet standards are disseminated as Internet RFCs.

DCE RFCs have no such necessary relationship with the OSF DCE offering (which is the moral equivalent of an Internet standard). Thus, for example, there is no commitment that components to be included in future releases of the DCE will be debated via DCE RFCs, nor is the DCE RFC series intended as an avenue for publication of the documentation for the DCE offering or for the DCE AES.

In the case of OSF and DCE, standardization would not be within the purview of the RFC process as it would violate pre-existing OSF processes for selection and standardization. This simplifies the RFC process considerably. All that is needed, therefore, is an RFC editor and an FTP archive site. The RFC editor should be limited to an administrative function: assigning numbers, copying into the archive and announcing availability. The OSF would support this function just as it presently deals with mailing lists and other such administrative tasks.

An RFC process is simply a tool for structured, informal conversations of technical concepts, ideas and proposed solutions. The DCE community stands to gain a great deal by adopting a DCE RFC process.

DCE RFCs must be submitted to the editor in markup language format. The editor also assigns the actual DCE RFC number. Internet RFCs are numbered with a single decimal number, and when they are reissued or updated, they acquire a new number. In contrast, DCE RFCs are numbered with a pair of decimal numbers, a *major* number *M* and a *minor* number *m* separated by a decimal point. This numbering syntax has the advantage of tracking the lineage of DCE RFCs more clearly.

Thus, the official names look something like *DCE-RFC M.m*. When they are reissued or updated, they (usually) retain the same major number, while the minor number is incremented: *DCE-RFC M.m+1*.

1.3.6 Available OSF Technologies

As an outcome of previous RFT processes, the following OSF technologies are available:

- ***The OSF/1 Operating System*** — The main technology providers are CMU (Carnegie Mellon University), IBM, Mentat, and NFS SecureWare. The AES for OSF/1 is available, and AIX for the RISC System/6000 conforms with it. The AES from OSF is defined as interfaces and services based on the precedence order of POSIX, ANSI C, XPG3, SVID 2, and BSD 4.3.
- ***The OSF/Motif Graphical User Interface on X-Windows*** — The main technology providers were HP and Digital. HP has been the main integrator. AES is available.
- ***The Common Desktop Environment (CDE)*** — This new GUI was originally developed by the COSE Consortium (Common Open Software Environment) consisting of IBM, Sun, HP, and Novell. The CDE maintenance and development of new versions was handed over to OSF and its PST process. COSE, with new members such as DEC, Fujitsu and Hitachi, will be working on new proposals for CDE (authoring phase of the PST process).
- ***The Architecture-Neutral Distribution Format (ANDF)*** — The submission came from the UK′s Defense Research Agency. ANDF enables software vendors to write and distribute a single version of an application that will run without any modifications or changes on PCs, workstations, minicomputers, and enterprise systems.
- ***The Distributed Computing Environment (DCE)*** — The main technology providers are: HP, Digital, SNI, and Transarc. IBM has been the main integrator. Work is going on to define the DCE AES.

1.4 OSF DCE Architecture

OSF DCE is a selection of different technologies submitted during the Request For Technology (RFT) issued by OSF in 1989 for the Distributed Computing Environment. DCE is a layer of services that allows distributed applications to communicate with a collection of computers, operating systems and networks. This collection of machines, operating systems and networks, when managed by a single set of DCE services, is referred to as a ***DCE cell***.

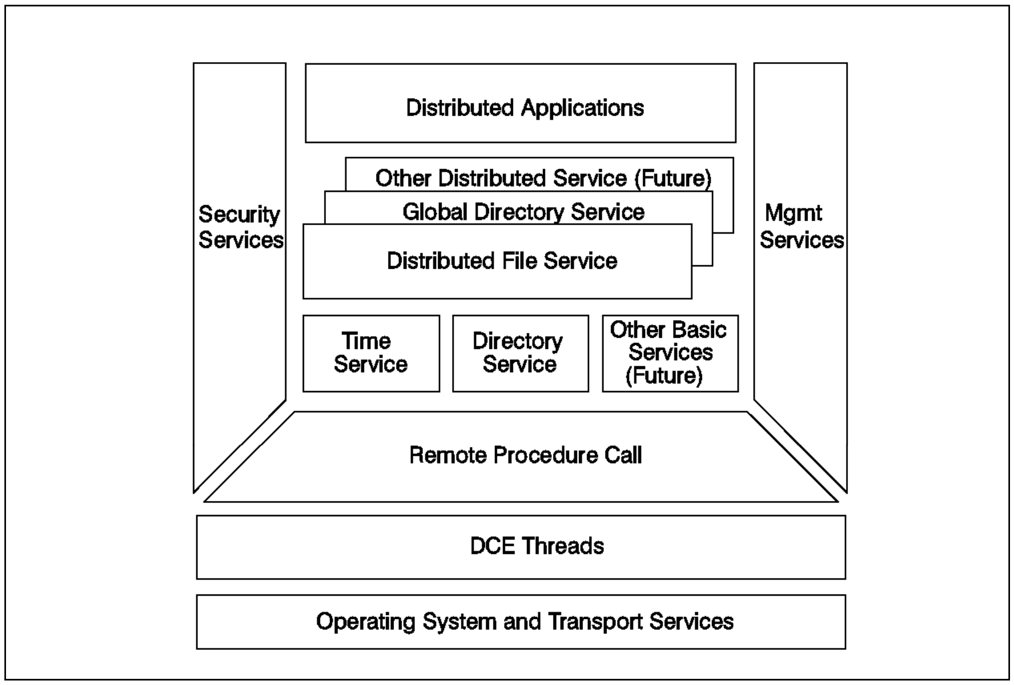


Figure 3. DCE Architecture

Figure 3 shows the relation of the DCE to distributed applications, network communications software and distributed (DCE) applications.

1.4.1 Models of Distributed Computing

OSF DCE components use three distributed computing models.

1.4.1.1 The Client/Server Model

In the client/server model, a distributed application is divided into two parts, *client* and *server*. In simple terms, the client is the entity that initiates the request for a service. The server is the entity that handles the request for a service.

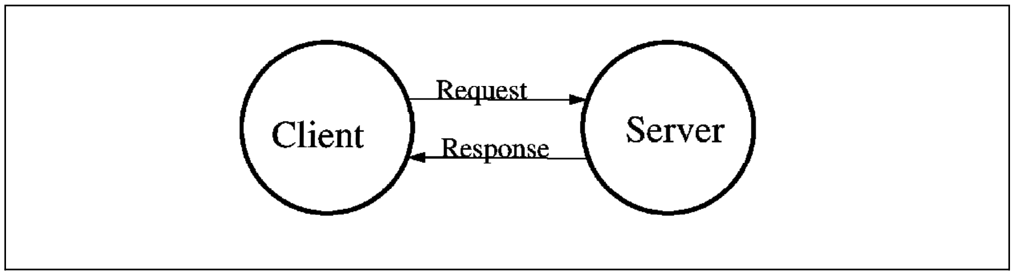


Figure 4. Client/Server Model

The terms client and server can refer to the role of a single application. For example, machine A may have a program that requests a piece of information from another machine, B. In this example, the program running on machine A is assuming the role of a client, while the program on machine B that fulfills the request is acting as the server. It is not hard to imagine that in a multitasking operating system environment we may have both client and server applications running on the same machine at the same time. It is also not hard to see that both the client and server functions for a transaction may both run on the same machine. In many cases, it will be necessary for the machine running the server to also run the client application in order to obtain access to the function it is serving. As you will see, this is the case with the OSF DCE core servers.

The terms client and server can also be used to describe the dedicated role of a given machine. This usually means that the service is available only from one machine or from a limited number of machines. For example, in a DCE network, one machine is designated to act as the directory server. Any machine that wishes to have access to the function provided by the Cell Directory Service (CDS) server process must also run the client program (CDS clerk). As a matter of fact, any machine that wants to become part of a DCE cell must run the DCE client (core) programs.

1.4.1.2 The Remote Procedure Call Model

In this model, the client makes what looks like a local procedure call. This procedure call is translated, and network communications are handled by the RPC mechanism. The server receives a request and executes the procedure, returning the results to the client. DCE RPC is an implementation of this model and is used by most of the other DCE technology components for their network communications.

1.4.1.3 The Data Sharing Model

While client/server and RPC are focused on distributed execution, *data sharing* is concerned with distributed data. In data sharing, the data of the server is sent to the client. Data sharing must address such needs as multiple copies of data, data consistency and managing simultaneous access to data. In OSF DCE, data sharing is built upon RPC, which is used as the means of transferring data. Both the Directory Service and the Distributed File System are based upon the data sharing model.

1.4.2 DCE Cell

The collection of machines that are managed together as a DCE unit is referred to as a *cell*. At a minimum, a cell must contain a Security Server, a Cell Directory Server and Distributed Time Servers. All of these services may run on one machine, or the servers can be spread among the machines that are to be part of the cell. The Directory, Time and Security Services are collectively known as the ***core services***.

1.4.3 DCE Security Service

Most multi-user operating systems provide some method to verify the identity of a user (authentication) and to determine whether a user should be granted access to a resource (authorization). In a distributed environment, a way has to be provided to authenticate requests made across the network and to authorize access to the network′s resources. There must also be a mechanism to protect network communications from attack. The challenge in a distributed environment is to provide these services transparently to both users and programs. For example, a user should not have to authenticate to each server in the network. The DCE Security Service can provide this level of functionality because of how it has been integrated with the other DCE services.

The DCE security specification was submitted by MIT. It is based on Kerberos Version 5.1 with some enhancements made by Hewlett-Packard Corporation. Kerberos is an authentication service that validates the identity of a user or service. The DCE Security Service is made up of several parts.

- The ***Authentication Service*** allows processes on different machines to determine each other′s identity (authenticate).
- The ***Privilege Service*** determines if an authenticated user is authorized to access a server resource. The Privilege Service provides information that servers need to determine the access that should be granted to the user.
- The ***Registry Service*** manages a security database used to hold entries for all principals. A *principal* is a user, server or computer that can communicate securely with another principal. The Registry Service is also used by administrators to maintain the list of principals known to DCE.
- The ***Audit Service*** detects and records security operations performed by DCE servers. This is new in OSF DCE 1.1.
- The ***Login Facility*** performs the initialization of the DCE environment for a user. It uses the Security Service to authenticate a user and returns credentials to the user. These credentials are then used to authenticate to other services in the DCE cell. The credentials expire after a set period of time or when the user exits from the DCE environment.

Most of these security components are transparent to the user.

1.4.4 DCE Directory Service

The Directory Service provides a naming model throughout the distributed environment that allows users to identify, by name, network resources, such as servers, users, files, disks, or print queues. The DCE Directory Service includes:

- Cell Directory Service (CDS)
- Global Directory Service (GDS)
- Global Directory Agent (GDA)
- Application Programming Interface (API)

The CDS manages information within a cell. The GDS is based on the CCITT X.500 name schema and provides the basis for a global namespace. The GDA is the CDS gateway to intercell communication. The GDA supports both Internet addresses and X.500 addresses. If the address passed to the GDA is an X.500 address, the GDA contacts the GDS. If the address passed to GDA is an Internet address, then the GDA uses the Internet Domain Name Service (DNS) to locate the foreign cell. Both CDS and GDS use the X/Open Directory Service (XDS) API as a programming interface.

1.4.5 DCE Distributed Time Service

Distributed Time Service (DTS) provides precise, fault-tolerant clock synchronization for the computers participating in a Distributed Computing Environment, both over LANs and WANs.

The synchronized clocks enable DCE applications to determine event sequencing, duration and scheduling. DTS is based on Universal Time Coordinated (UTC) time, an international time standard. The specification of the Time Service was submitted by Digital (DEC), and it is based on DEC Distributed Time Synchronization. DTS will be discussed in greater detail in Chapter 4, “Distributed Time Service” on page 79.

1.4.6 Distributed File System

The Distributed File System (DFS) presents directories and files in a global namespace that can be accessed from any DFS client. Caching on DFS clients reduces access time and network traffic and results in high performance.

DFS includes support for both Journaled File System (JFS) and Local File System (LFS) formats. LFS is a fast-restarting, log-based physical file system that supports file replication for high availability.

DFS files and directories can be protected by using Access Control Lists (ACL). You can define an ACL for each file or directory to restrict or authorize access. With DFS ACLs, you have the granularity to control access for users and groups of the local or any foreign cell. DFS ACLs are different than the access control list support provided through the AIX operating system.

The DFS is built on top of the core technologies: Security Service, Cell Directory Service and Distributed Time Service. DFS also makes use of threads and RPCs. It implements a superset of the POSIX 1003.1 file system semantic standard submitted by Transarc Corporation and is based on the Andrew File System. DFS will be discussed in Chapter 5, “Distributed File Service” on page 97.

1.4.7 Threads

Threads support the creation, management and synchronization of multiple paths of control within a single process. Threads specification was submitted by DEC to the Open Software Foundation; it is based on DEC′s Concert Multithread Architecture (CMA). The threads programming facility is also POSIX 1003.4a Draft 4 compliant. If threads are already available on the operating system, DCE can use them. Threads will be discussed in Chapter 11, “Threads” on page 209.

1.4.8 Remote Procedure Call

This is a complete environment to help you develop client/server applications. A development tool, consisting of an Interface Definition Language (IDL), is provided. The RPC runtime service facilitates the implementation of the network protocols used by the client and server applications to communicate. One component of the RPC is the *uuidgen*, a program that generates a Universally Unique Identifier (UUID; a 32-digit number) to uniquely identify resources, services and users in DCE, independently from time and space. The RPC specification was submitted by Apollo/HP, and it is based on Network Computing System architecture (NCS) Release 2. RPC will be discussed in greater detail in Chapter 10, “Remote Procedure Calls” on page 173.

1.5 OSF DCE 1.1 New Features

OSF DCE Version 1.1 offers many enhancements over OSF DCE Version 1.0.3. Following is a summarized list of the enhancements and improvements that have been incorporated into OSF DCE Version 1.1.

1.5.1 Improved Administrative Functions

- ***DCE Control Program (dcecp)*** — Provides a single administrative interface to many of the DCE administrative functions. The dcecp interface includes Tcl, a powerful scripting language that can be used to customize and simplify many administrative tasks.
- ***DCE Daemon (dced)*** — Enables remote configuration and administration of DCE services. Administrative highlights include startup, shutdown, status queries, and cell configuration information.
- ***Enhanced diagnostic messaging capability*** — Allows instrumenting services to capture more information and unify the message format across all DCE components.
- ***Cell Aliasing*** — Permits a cell to have multiple names and allows the primary name of a cell to be changed.
- ***Hierarchical Cells*** — Allows cell names to be registered in the CDS and allows multiple cells to be organized to reflect the hierarchical structure of an organization. This feature will initially be missing in IBM DCE for AIX Version 2.1.

1.5.2 Security Improvements

- ***Security Delegation*** — Allows intermediary servers to act on behalf of an initiating client while preserving the client′s and server′s identities and access control attributes across chained RPC operations.
- ***Auditing*** — Allows administrators to track security-related events within DCE′s trusted computing base. An API is included and permits development of servers that record audit events, or it can be used to create tools that can analyze audit records.
- ***Generic Security Service Application Program Interface (GSS-API)*** — Allows non-RPC applications to use DCE security features. GDS has been extended to use DCE security via the GSS-API.
- ***Extended Registry Attributes (ERA)*** — Enables single sign-on across non-UNIX platforms and legacy applications by providing a secure way of associating additional security information with users and groups.
- ***Extended Login Capabilities*** — Provides pre-authentication, password management and enables applications to require access only from trusted machines.
- ***ACL Manager Library*** — Supports easier implementation of access control list (ACL) managers for application servers.
- ***Group Override*** — Customizes the group name mapping from host to host to allow DCE to adopt to various operating system conventions.

1.5.3 Internationalization

- ***Internationalized Interfaces*** — Allows the use of message catalogs for all user-visible messages. It is now possible to *localize* DCE programs by supplying DCE messages in other languages.
- ***Character Code Set Interoperability*** — Allows development of RPC applications which automatically convert character data from one code set to another.

1.5.4 Performance Enhancements

- ***IDL Compiler*** — Generates smaller, cleaner RPC stub code and supports a number of new IDL constructs, such as unique pointers, user exceptions and node deletions.
- ***RPC Throughput Enhancements*** — Are provided by access to additional sockets for peak usage and optimization of RPC runtime packets for transmission and fast transport, such as FDDI or satellite.

1.5.5 Other Enhancements

- Modifications to various GDS components.
- DFS-NFS Gateway now supports PC-NFS.
- DFS Delegation allows a file to be passed with the initiator′s privileges intact.
- Subtree operations allow large-scale administrative name changes in a cell.

1.6 IBM Added-Value Components for DCE

IBM has added several components to the base OSF DCE offering to provide the following functions:

- DFS access from NFS clients
- Management functions from SystemView
- Exportable data encryption
- Online documentation

1.6.1 DCE NFS to DFS Authenticating Gateway for AIX

Many customers use Network File System (NFS) technology to distribute file systems over a network. The NFS/DFS gateway is a product on the AIX platform that allows NFS client systems to access the DFS file space. The NFS client is typically run on a system that is not part of the DCE cell or one that has no DFS code installed.

The NFS/DFS Gateway is installed on a DFS client system that exports its DFS file system into NFS, thus acting as an NFS server. The gateway provides a bridge between the authentication methods of DFS and NFS. This is accomplished by connecting an NFS client with a DCE principal. The NFS/DFS Gateway allows the NFS client to obtain authenticated access to the DFS file space.

The NFS/DFS Authenticating Gateway is available on DCE Version 1.3 (AIX 3.2.5) and DCE Version 2.1 (AIX 4.1.3). The new version (AIX DCE 2.1) supports automated authentication from PC-NFS clients.

1.6.2 DCE Manager for AIX

DCE Cell Manager is a product that allows NetView for AIX to automatically discover and monitor all DCE core servers (Security, Time, Directory, RPC, Global Directory). DCE Cell Manager also permits monitoring and discovery of all DFS servers. DCE Cell Manager uses NetView for AIX′s object and topology databases to store its data. It can be launched from NetView for AIX and makes use of the NetView online help facility.

The DCE Manager is currently only available on DCE 1.3 for AIX 3.2.5.

1.6.3 User Data Masking Facility

The RPC communication provides different security levels, the highest being full data encryption. However, the DES algorithm (data encryption standard) internally used by DCE cannot be exported outside the U.S. in a user accessible form. This means it cannot be used for data encryption.

On the AIX and OS/2 platforms, there is a User Data Masking Facility, which is still referred to as Common Data Masking Facility or CDMF. CDMF allows you to encrypt user data in RPCs using DES with a 40-bit key instead of the standard 52-bit key. Since this makes the encryption weaker, it has less export restrictions from the U.S. It is a good solution for non-U.S. customers who want increased privacy, but cannot have an export license for full DES.

1.6.4 Online Documentation

All DCE manuals are provided in softcopy form to be accessed with a graphical viewer. The graphical softcopy files are INF files, which is the standard format for OS/2 online documentation. They can be accessed through the Interactive Presentation Facility (IPF).

The IBM DCE Version 2.1 for AIX Version 4.1 provides an IPF viewer for X-Windows (IPF/X). The `xview` command that starts IPF/X provides hypertext linking, search and print facilities, inline graphics display, a bookmark function, and online help. Its startup is integrated into InfoExplorer. IBM DCE 2.1 for AIX also provides the documentation in ASCII from which can be viewed from ASCII terminals with an ASCII browser. The `dceman` command emulates MAN pages for DCE commands.

On IBM DCE 1.3 for AIX, softcopy documentation is in InfoExplorer format.

1.7 IBM DCE Product Information

This section summarizes the DCE functions supported on the AIX and OS/2 platforms as well as the product packaging and system requirements. We provide also information about the current DCE offering on the DOS Windows platform.

DCE Products on Non-IBM Platforms

A complete listing of products that support the DCE environment is available on the Internet at:

`http://www.osf.org/comm/lit/dce-prod-cat/`

For a DCE function summary on all IBM platforms, see 1.7.4, "IBM DCE Cross Platform Matrix 9/95" on page 18 at the end of this section.

1.7.1 IBM DCE for AIX

The RISC System/6000 currently supports two major versions of AIX and DCE:

- AIX DCE Version 1.3 for AIX Version 3.2.5 at OSF DCE Level 1.0.3
- AIX DCE Version 2.1 for AIX Version 4.1.3 at OSF DCE Level 1.1

The two DCE versions can perfectly coexist and interoperate in the same DCE cell but can only run on the specified version of AIX.

Table 1. AIX DCE Function Summary

DCE Function	AIX DCE 1.3	AIX DCE 2.1
OSF DCE Level	1.0.3	1.1
Threads	√	√
RPC Client and Server	√	√
Time Service (DTS) Clerk and Server	√	√
Security Service Client and Server	√	√
Directory Service (CDS) Client and Server	√	√
Global Directory Agent (GDA)	√	√
Global Directory Service (GDS) Client and Server	√	—*
DFS Client	√	√
Enhanced DFS Server	√	√
Note: (*) The Global Directory Service is supported with AIX DCE 1.3		

The ***product packaging*** has considerably changed between the two versions. The AIX DCE 2.1 bundles all base services with AIX. So, the following features are part of AIX Version 4.1.3:

- Threads (DCE Threads compatibility library) and RPC
- All client functions (CDS, Security, DFS)
- DTS server
- Base DFS server (no Local File System support)

Table 2. IBM Products for AIX DCE Version 1.3

Product Number	Description
5765-232	IBM DCE Threads for AIX (Version 1.3) *
5765-117	IBM DCE Base Services for AIX (Version 1.3) (includes all clients) **
5765-118	IBM DCE Security Server for AIX (Version 1.3)
5765-119	IBM DCE Cell Directory Server for AIX (Version 1.3)
5765-121	IBM DCE Enhanced Distributed File System for AIX (Version 1.3)
5765-120	IBM DCE Global Directory Server for AIX (Version 1.3)
5765-259	IBM DCE Global Directory Client for AIX (Version 1.3)
5765-456	IBM DCE Manager for AIX (Version 1.3)
5765-457	IBM DCE NFS to DFS Authenticating Gateway for AIX (Version 1.3)
Note: (*) The Threads package is included in DCE Base Services for AIX. This separate offering enables the usage of threads with AIX 3.2.5 in an environment without DCE. (**) The DCE DFS Base Services are included in the DCE Base Services.	

Table 2 above lists the packaging with product numbers for AIX DCE 1.3, whereas Table 3 on page 16 does the same for AIX DCE 2.1.

Table 3. IBM Products for AIX DCE Version 2.1

Product Number	Description
5765-533	IBM DCE Security Services for AIX (Version 2.1)
5765-534	IBM DCE Cell Directory Services for AIX (Version 2.1)
5765-537	IBM DCE Enhanced Distributed File System for AIX (Version 2.1)
5765-540	IBM DCE NFS to DFS Authenticating Gateway for AIX (Version 2.1) **
5765-532	IBM Getting Started with DCE for Application Developers (Version 2.1) *
5765-538	IBM DCE User Data Masking Encryption Facility for AIX (Version 2.1) **

Note:

The DCE/DFS Base Services for AIX (Version 2.1) are now included in the AIX operating system version 4.1.3 (5765-393).

(*) This is a combined software/service offering that includes DCE tools, attendance in one LAN Systems Workshop and a one year subscription to the IBM Developer Connection for AIX program.

(**) See 1.6, "IBM Added-Value Components for DCE" on page 13 for more information

The ***system requirements*** for a DCE/DFS client machine is at least 16 MB of RAM and 35 MB of harddisk, including a 10 MB cache for DFS. If online publications are to be loaded on a local disk, add another 15 MB of harddisk space. Systems running DCE/DFS servers should at least have 32 MB of RAM. Their harddisk requirement depends on the amount of data they need to store.

1.7.2 IBM DCE for OS/2 Warp

The OS/2 platform currently supports a product version of DCE at OSF level 1.0.2 and a beta version of OFS DCE 1.1, which will become a product in the near future:

- OS/2 DCE Version 1.2 for OS/2 Version 2.1 at OSF DCE Level 1.0.2
- OS/2 DCE Version 2.1 for OS/2 Warp Beta at OSF DCE Level 1.1

Table 4. OS/2 DCE Function Summary

DCE Function	OS/2 DCE 1.2	OS/2 DCE 2.1
OSF DCE Level	1.0.2	1.1
Threads	√	√
RPC Client and Server	√	√
Time Service (DTS) Clerk and Server	√	√
Security Service Client and Server	√	√
Directory Service (CDS) Client and Server	√	√
Global Directory Agent	—	√
User Data Masking	—	√
DFS Client	—	√

The ***product packaging*** information for OS/2 DCE Version 2.1 is not available yet. Table 5 on page 17 below lists the packaging with product numbers for the currently available product OS/2 DCE 1.2, which may soon be outdated.

Table 5. IBM DCE Products for OS/2 Version 1.2

Product Number	**Description**
5696-657	IBM DCE SDK for OS/2 and Windows Version 1.0
5696-692	IBM DCE Client for OS/2 Version 1.0

The specifications that go with the beta version list the ***system requirements*** as shown in Table 6 below.

Table 6. OS/2 DCE Minimum and Recommended Hardware Configurations

Workstation Type	**Minimum Configuration**	**Recommended Configuration**
DCE Client	• 25 MHz 386 System • 12 MB RAM • +4 MB RAM for DFS	• 33 MHz 486 System • 16 - 24 MB RAM
DCE Server	• 33 MHz 486 System • 16 MB RAM (with one core server) • 20 MB RAM (for two core servers)	• 50 MHz 486 System • 24 MB RAM, or more

1.7.3 IBM DCE for DOS Windows

The current DCE implementation on DOS Windows is based on OSF DCE 1.0.2 and originally developed by Gradient Technologies. It provides full client and server support for RPC as well as client support for CDS, security service and DTS.

IBM has plans to bring enhancements to their DCE for Windows. Several vendors will also come up with implementations of OSF DCE 1.1 and a DFS client.

The following table lists the order numbers for the currently available IBM DCE for DOS Windows product:

Table 7. IBM DCE Products for Windows

Product Number	**Description**
5696-657	IBM DCE SDK for OS/2 and Windows Version 1.0
5696-690	IBM DCE Client for Windows Version 1.0

IBM DCE for Windows runs on any Intel platform with a 386 processor or higher, at least 4 MB of memory, 5 MB of free disk space, and 5 MB of Windows swapfile. It requires DOS 5.0 or higher and Windows 3.1.

1.7.4 IBM DCE Cross Platform Matrix 9/95

Table 8. IBM DCE Product/Function Matrix

DCE Function	**OS/2 Warp**	**Windows**	**AIX 3.2**	**AIX 4.1**	**OS/400**	**MVS/ESA**	**VM/ESA**
OSF Level	1.1 Beta	1.0.2	1.0.3	1.1	1.0.2	1.0.2	1.0.2
DCE Core Client Functions							
Threads	√	√	√	√	√	√	Ann
RPC Client	√	√	√	√	√	√	Ann
CDS Client	√	√	√	√	√	√	Ann
DTS Clerk	√	√	√	√	√	√	SoD
Security Service Client	√	√	√	√	√	√	Ann
DCE Core Server Functions							
RPC Server	√	√	√	√	√	√	Ann
DTS Server	√		√	√	SoD	√	SoD
CDS Server	√		√	√	SoD		
Security Server	√		√	√	SoD	SoD	
X.500 Inter-Cell Access							
Global Directory Service (Client)			√				
Global Directory Service (Server)			√				
Data Sharing Services							
Distributed File System Client	√	OEM	√	√	SoD	Beta	
Enhanced DFS Server			√	√		√ (1)	
Other Services							
Encina Clients	√	OEM	√	√			
Encina Servers			√	√			
Application Support IMS, CICS	N/A	N/A	N/A	N/A	N/A	√	N/A

Note:

√	Implemented today.
(1)	OSF Level 1.0.3. No FLDB or Backup server available; an AIX system must provide these services.
OEM	Available from an Independent Software Vendor, such as Transarc or Gradient.
Beta	Early code customer test (beta).
SoD	Statement of Direction.
N/A	This feature is not applicable to this platform.
Ann	Feature has been announced, the General Availability date is February 23, 1996.

Chapter 2. Directory Service

A distributed computing environment contains many users, computers, applications, and printers dispersed in the network. This creates a complex group of resources and users that somehow have to be located. These resources and users, also referred to as objects, can be easily located if we have a centralized process that keeps track of every change in the network.

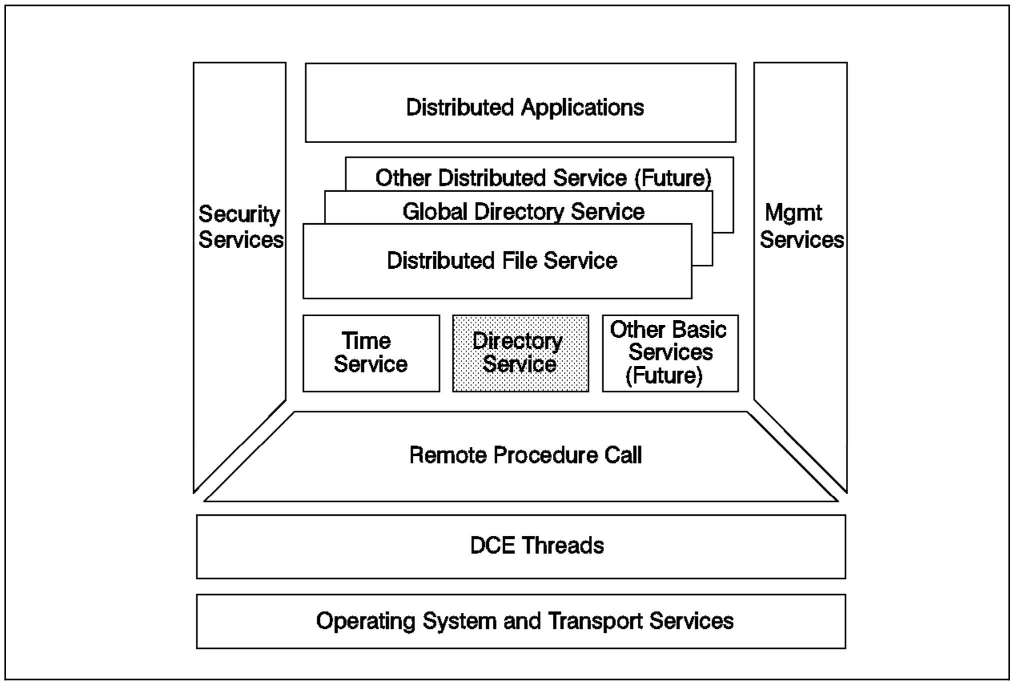


Figure 5. OSF DCE Directory Service

The directory service is the process that makes it possible for the user to locate objects in the network without knowing their physical location. It hides from the user the distributed nature of the environment. It is like a telephone directory assistance service that provides the phone number when given a person′s name.

Users do not normally access or use the directory services directly. They run applications which might use the directory services to find objects. The only thing a user might have to know is object names and maybe the naming model.

DCE administrators must understand the directory service to be able to administer objects and manage the service and its database.

A programmer can use the X/Open Directory Service (XDS) API to directly access the directory service or the RPC Name Service Interface (NSI) from within DCE RPC applications.

This chapter will describe the concepts related to the directory service and the way it manages the naming environment in DCE.

2.1 What is a DCE Cell?

A cell is a group of users, systems and resources that are typically centered around a common purpose and that share common DCE services. The number of systems and physical locations does not define the cell boundaries, but it is influenced by:

- Business Needs — Different groups in an organization, such as marketing, manufacturing or development, may want to share different resources. The number of objects that require shared access, and the frequency of shared access can determine the boundaries of a cell.
- Administration — The number of administrative tasks related to DCE services and their respective databases as well as the number of people available to do these tasks can also determine the boundaries of a cell. Many small cells require administration of many DCE core servers and intercell setup. Intercell setup is not just defining GDS or DNS, it also means management of permissions for foreign users. This is what may cause a tremendous administration overhead.
- Security — Security in a cell is a big issue, and it can be interesting for the organization to define more cells to reduce the risk of break-ins or the amount of work necessary to recover from a break-in.
- Overhead — Usually, there is more interaction within a cell; so the boundaries have to be defined considering the kind of resources the users need to access and how often they access them.

There is not a clear recipe for a company to decide whether it is going to have one (large) or multiple (smaller) cells. This decision must be made on a per-company basis and includes a lot of decision factors, such as the ones listed above.

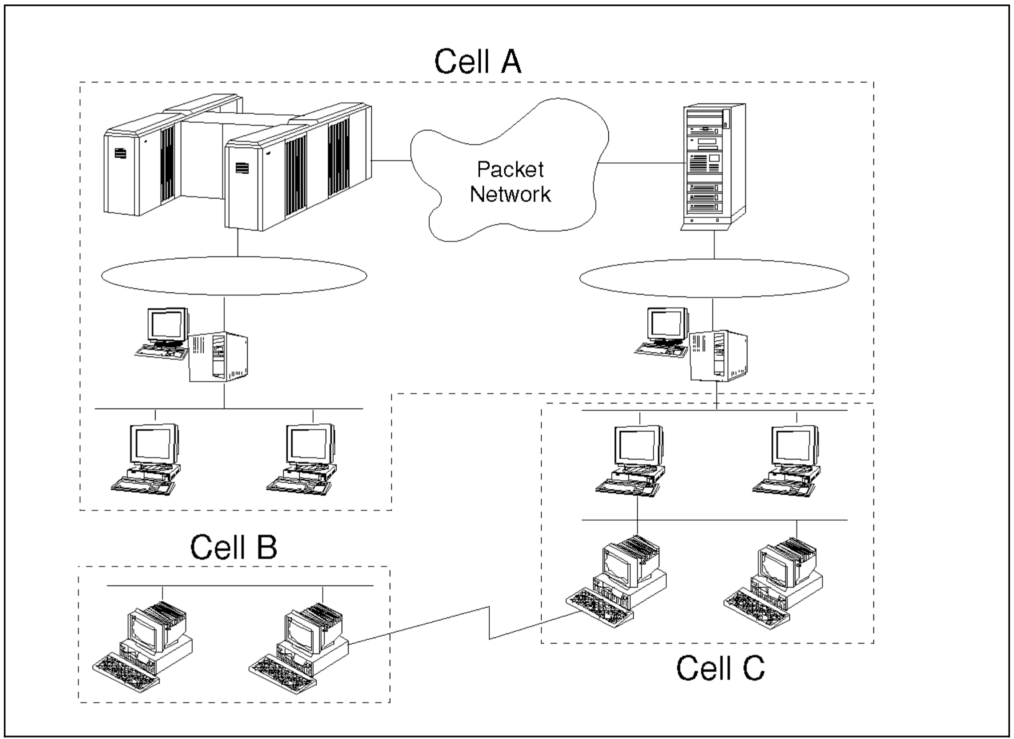


Figure 6. Multi-Cell Environment

Figure 6 gives an example of a multi-cell environment. As you can see, cell A includes a complex network including different LANs and WANs. Cell B is only a LAN, and cell C includes two interconnected LANs.

Each cell is a self-sufficient, independently managed unit in a global distributed computing environment. It must at least have the following DCE core services:

- One Security Server
- One CDS Server
- Three DTS Servers per LAN (the use of DTS is optional)

2.2 Directory Services Component Overview

The directory service component that controls names inside a cell is called the Cell Directory Service (CDS). The CDS stores names of resources in that cell so that when given a name, CDS returns the network address of the named resource.

Sometimes we need to find resources outside the cell. The directory service component that helps resolve foreign names is called Global Directory Service (GDS). Access to GDS is through an intermediate component called the Global Directory Agent (GDA). This is called a two-tier architecture.

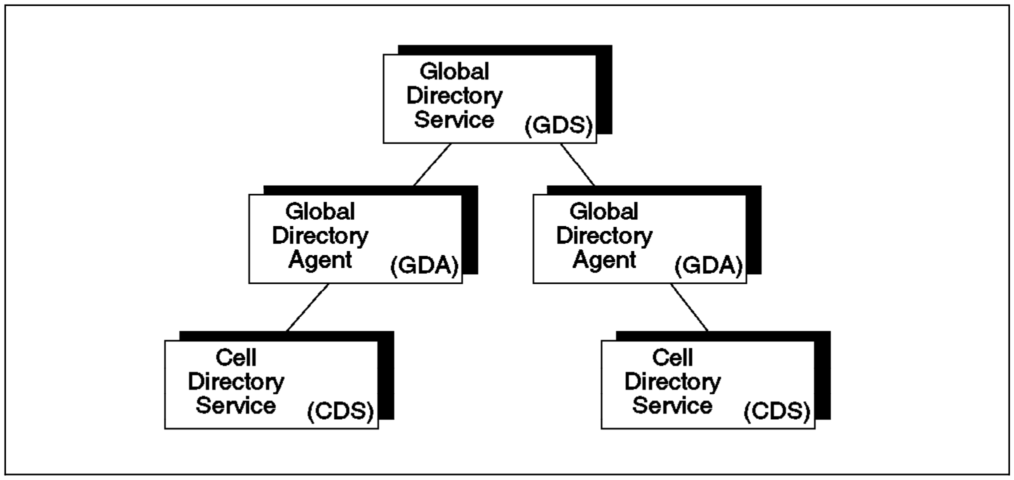

Global Directory Service (GDS)
Global Directory Agent (GDA)
Global Directory Agent (GDA)
Cell Directory Service (CDS)
Cell Directory Service (CDS)

Figure 7. Components of the Directory Service

The origin of the CDS is Digital Equipment′s Distributed Naming Service (DECdns), and the origin of the GDS is the Siemens Dir-X implementation of the CCITT X.500/ISO 9594 international standard. X.500 is an emerging global directory service standard, but the Internet domain name system (DNS) is an established industry standard. For interoperability purposes, GDS supports both X.500 and DNS transparently.

Note: The Global Directory Service (GDS) is not provided in the AIX DCE 2.1 release nor in DCE for OS/2. However, GDS can exist in the same cell and can be used for intercell communication if it is provided by an another product, such as AIX DCE 1.3.

2.3 The DCE Global Naming Environment

DCE Naming Service provides a naming model throughout the distributed environment. This model allows users to identify, by name, resources, such as servers, files, disks, or print queues, and gain access to them without needing to know where they are located in a network. Further, users can continue referring to a resource by the same name even when a characteristic of the resource changes, such as its network address.

In this section, we will explain the structure of this global namespace and how names are built to become unique and addressable throughout the whole world.

2.3.1 The Global Name Space

The global distributed computing environment is composed of administratively independent cells. The name space is hierarchically organized and forms a tree, with containers (directories) and leaf objects. As illustrated in Figure 8 below, the root directory /... is global and contains all cell names on its first subdirectory level.

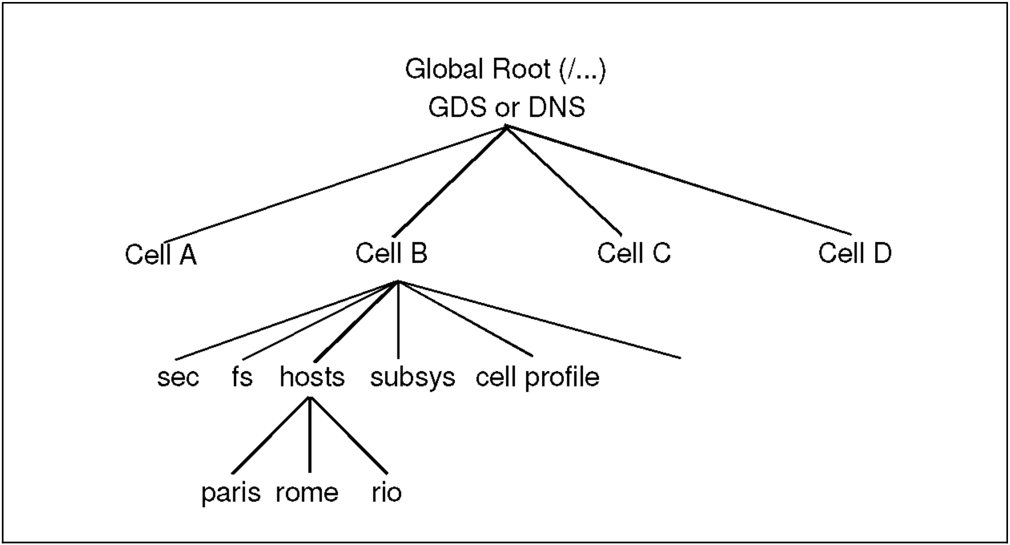


Figure 8. The Global Name Space with Independent Cells

The cell names build the root directories for each cell. The subtrees underneath each /.../<cellname> directory are under the management domain of their respective cell. However, the basic subtree structure of each cell is the same, so that users can rely on something fixed when accessing foreign cells. We will explain the cell namespace in 2.3.5, "The DCE Cell Namespace" on page 27.

Users defined in Cell A can access objects in Cell B using the ***full name*** for objects, such as */.../CellB/cell-profile*. Users in Cell B can, of course, do the same. However, since most access is supposedly in their own cell, users in Cell B can use an abbreviated ***local name*** to access all objects in the local Cell B. The local name is */.:/lan-profile*.

2.3.2 Cell Names

To be globally addressable, cell names must be unique. There must be an administration authority that keeps track of names and assigns new, unique names. Furthermore, there must be some global network routing mechanism that can find a communication path to the requested cell so that a foreign cell can be accessed.

There are two well-established naming schemes in place that DCE makes use of:

- CCITT X.500
- Internet Domain Name Service (DNS)

The only well-established, multi-vendor-supported global network today is the Internet. It has global addressing and routing. The DNS naming scheme makes direct use of the Internet naming and routing scheme by extending the information that each Internet DNS server carries. The X.500 naming scheme is independent from the Internet and more general. It is implemented with the Global Directory Service (GDS), which can store any kind of object. DCE uses GDS to store cell names and their addresses, which today are also Internet addresses. So, the access of the foreign cell is established over the Internet in both cases.

2.3.2.1 X.500 Names

Figure 9 shows a global name that refers to a printer queue object defined in the IBM ITSO cell. Local users can address it with /.:/susbsys/PrintQ. The prefix (/...) indicates that the name is global. Following the prefix, the X.500 syntax defines four blocks, each one with two parts separated by an equal sign (=). The abbreviation of each block stands for country (C), organization (O), organizational unit (OU), and common name (CN, not shown).

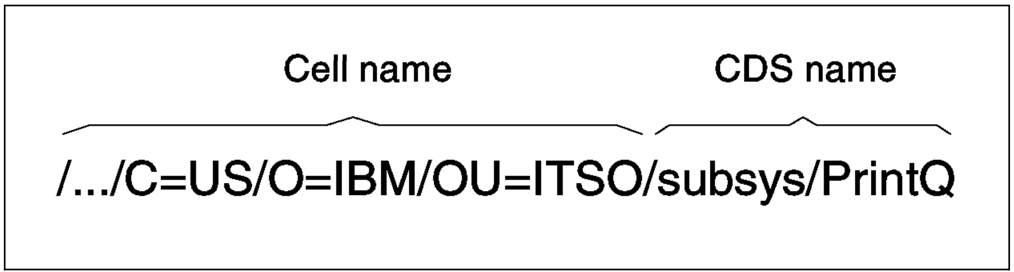


Figure 9. Global Representation of a Subsystem Printer Queue

2.3.2.2 DNS Names

The DCE naming environment supports DNS based on the Internet RFCs 1034 and 1035. The DNS is very common in many networks as a name service for host names. It can also be represented as a hierarchical tree with its top-most levels under the control of the Network Information Center (NIC).

The name directly under the root is a two-letter code for country (such as *us* or *uk*) as defined in ISO standard 3166. 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.

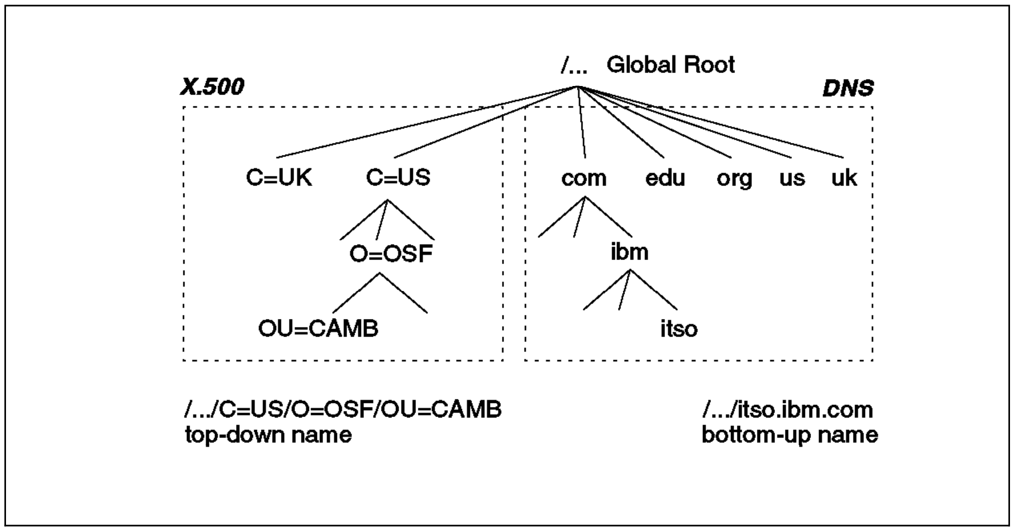


Figure 10. Comparison of Cell Name Representations

Figure 10 shows a comparison of the DNS hierarchical tree structure and the X.500 CDS representation. X.500 picks the names in a top-down order, while DNS does it in bottom-up order.

2.3.2.3 Cell-Name Aliases

Cell-name aliasing is a new function in the OSF DCE 1.1 release. Cell aliasing enables cell names to be changed and allows cells to have multiple names to reflect changes in an organization. Your cell has a *primary name*, which is the name that DCE services return for the cell when queried, and one or more *alias names* that the DCE services recognize in addition to the primary name. For example, if your cell is registered in the GDS global directory service, and you want to register it in the DNS as well, you obtain a DNS name for the cell, and set it up as a cell alias. The GDS name remains the primary name.

To change the cell name, you would first assign an alias name with the following command:

```
dcecp> cellalias create <new_name>
```

Then you can make the alias name the new primary name with the following command:

```
dcecp> cellalias set <new_name>
```

This step contacts every DCE host daemon in the cell and makes it interchange the values stored for the cell name and cell aliases.

Note: Cell aliasing is not implemented in AIX DCE 2.1; it will be supported in the next version.

2.3.3 Hierarchical Cells

A new function in OSF DCE Release 1.1 are the hierarchical cells. Independent DCE cells can now be connected into a hierarchical cell configuration which makes it easier for companies to build cell structures that correspond to the company structure. Cell names can now be registered in CDS, thus making one cell′s CDS the higher-level directory service and the cell itself the parent cell.

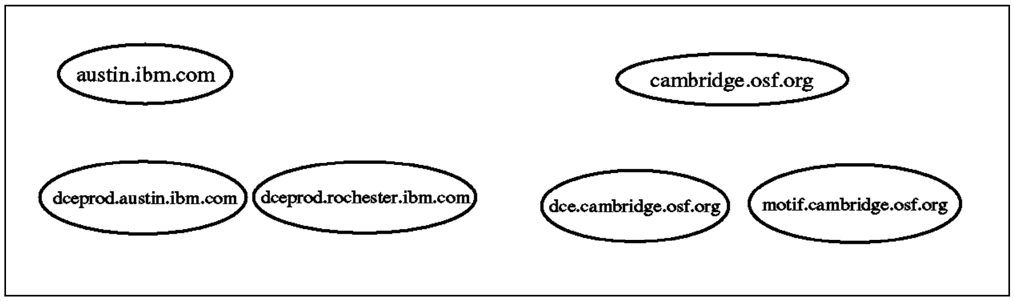


Figure 11. Related Cells

Figure 11 shows DCE cells that may have grown independently from each other. They may have set up intercell communication individually and exported their names into DNS (or GDS if the cell names were X.500). These cells may now be integrated into hierarchical structures within their companies with only the top-most cell names exported into DNS.

A child′s cell name consists of its parent′s cell name and an additional child name in CDS syntax. The parent cell name can be queried with the `getcellname` command. The child cell has to establish its new name as an alias name and make it the primary name. It then uses a `dcecp registry connect` command to create the cross-cell authentication information with the parent cell and a `dcecp cellalias connect` command to register its cell name with the parent cell′s CDS server(s).

The hierarchy can include multiple levels. Figure 12 illustrates the cell hierarchy that results from restructuring the cells of Figure 11. Child cell names look like regular CDS objects of the parent cell.

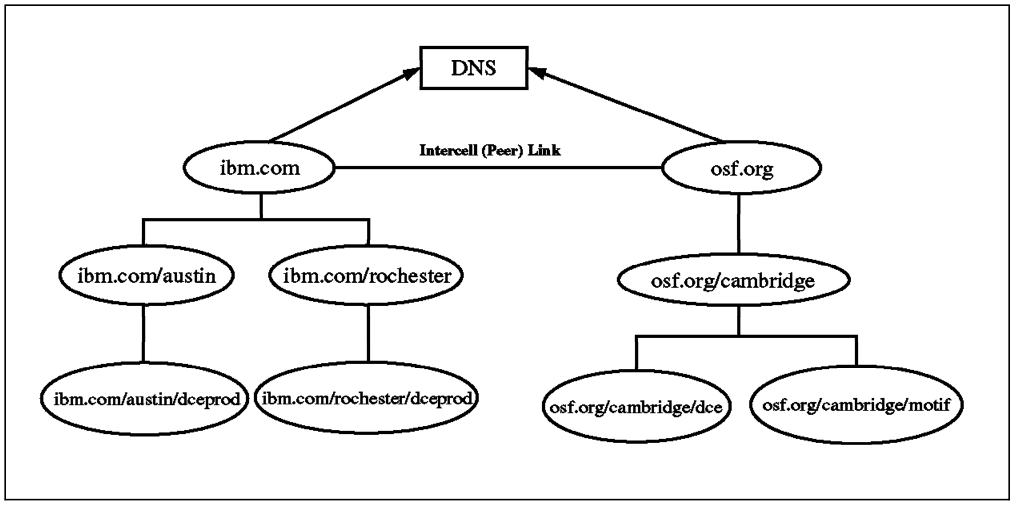


Figure 12. Hierarchical Cells

The term *hierarchical cells* might be misleading. Child cells are full-fledged, self-sufficient cells, and they have their own security servers and everything needed in a cell. Parent and children are not one big cell with shared resources; they are still foreign cells to each other. What is actually hierarchical is the way cell names and intercell communication definitions are administered.

The advantage, for the time being, is that intercell communication can now be defined within CDS. Global naming services (GDS or DNS) need not be involved.

Part of the concept, but not implemented in OSF DCE 1.1, is the concept of transitive trust, which will make security management across cells easier.

Note: Hierarchical cells are not implemented in AIX DCE 2.1; this will be supported in the next version.

2.3.4 Intercell Routing Services

When the directory service of the local cell (Cell Directory Service or CDS) receives a global name that starts with global root (/...), it determines that it is a foreign name, unless the cell name is equal to its own cell name.

If it is a foreign name, the CDS server returns the address of a Global Directory Agent (GDA) contained in the CDS_GDAPointers attribute of its root directory. The CDS clerk sends the request to the GDA. When the request is for a hierarchical cell name, the GDA passes it to the CDS server of the top-most cell in the hierarchy. The CDS server in this cell walks down the cell hierarchy to locate the name. This is the concept of ***hierarchical cells*** introduced with OSF DCE 1.1.

If the request is for a foreign cell outside the same hierarchy, the GDA sends the request to a global service and returns the address of the CDS server that contains the requested object. This is illustrated in Figure 13 below.

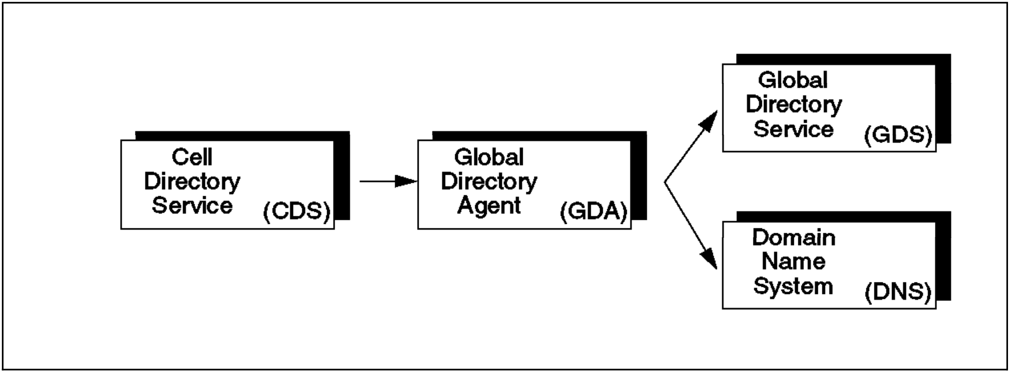

Cell Directory Service (CDS)
Global Directory Agent (GDA)
Global Directory Service (GDS)
Domain Name System (DNS)

Figure 13. Name Search Routing between CDS, GDA and GDS

The GDA recognizes the type of name that is used. If it receives an X.500 cell name, it passes the request to the Global Directory Service (GDS) client in its own cell. The GDS client passes the request to a GDS server, which can be anywhere in the whole global network. If the name is a DNS name, the GDA passes the request to its local DNS server to resolve the address of the foreign cell.

The result of the look-up is the address (binding information) to a CDS server in the foreign cell. The CDS client that made the request then must connect to the foreign CDS server to obtain the information as if the CDS server were in the local cell. See 2.4, "Cell Directory Service (CDS)" on page 29 for explanations on how look-ups are performed in a cell.

6.3, "Setting Up Intercell Communication" on page 131 provides guidance on how to set up intercell communications.

2.3.5 The DCE Cell Namespace

Figure 14 shows the DCE cell namespace created by CDS for each cell in the DCE environment. Since the CDS namespace is independently managed within the DCE cell, an administrator can add to this structure or even change it. However, this basic structure should be maintained in every cell so that interoperability is easier.

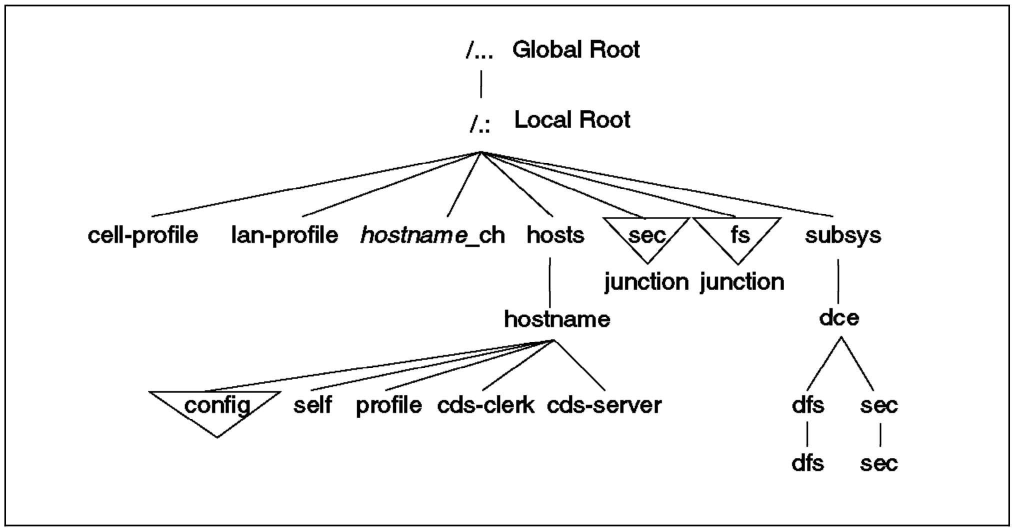


Figure 14. Directories Created for Each Cell

Not all DCE names are stored directly in the DCE directory service. Some services, such as the Security Service (*sec*) and the Distributed File System (*fs*), connect into the namespace by means of specialized CDS entries, called ***junctions***. A junction entry contains binding information that enables a client to connect to a directory server outside of the directory service.

In DCE 1.1, the DCE host daemon on every node also manages a part of the namespace, starting from the */.:/hosts/<hostname>/config* junction of every host. DCE daemon objects, such as keytab objects managed in the */.:/hosts/<hostname>/config/keytab* directory, allow local resources on the *<hostname>* host to be remotely managed from other nodes in the cell. More information on this topic can be found in 3.2.3.6, "Keys and Key Management" on page 57.

The security namespace is managed by the registry service of the DCE security component, and the DFS namespace is managed by the Fileset Location Database (FLDB) service of DFS.

The following list contains a description of the different standard entries:

- **/.:/cell-profile** - This is the default master profile for the cell where all hosts, users and other profiles must chain up to.
- **/.:/lan-profile** - This is the default LAN profile used by DTS to store the names of the local DTS server set.
- **/.:/***hostname***_ch** - This is the clearinghouse of the cell.
- **/.:/hosts** - This is where host directories are catalogued.
- **/.:/hosts/***hostname* - Each host has a directory in which RPC server entries, groups, and profiles associated with this host are stored.

- **/.:/hosts/***hostname***/config** - This entry contains binding information to the various *dced* objects, such as keytab or hostdata. For every DCE host, this is a junction to the dced-administered namespace.
- **/.:/hosts/***hostname***/self** - This entry contains a binding to the *dced* daemon on host *hostname*.
- **/.:/hosts/***hostname***/profile** - This is the default profile for host *hostname* and it must contain a default which points at /.:/cell-profile.
- **/.:/hosts/***hostname***/cds-clerk** - This entry contains the binding for a CDS clerk.
- **/.:/hosts/***hostname***/cds-server** - This entry contains the binding for a CDS server.
- **/.:/sec** - This is the RPC group of all security servers for this cell.
- **/.:/fs** - This is the RPC binding of all fileset database machines housing the Fileset Location Database (FLDB).
- **/.:/subsys** - This directory contains directories for different subsystems in this cell.
- **/.:/subsys/dce** - This directory contains DCE specific names.
- **/.:/subsys/dce/dfs** - This directory contains DFS specific names.
- **/.:/subsys/dce/dfs/bak** - This entry contains the RPC bindings of all. backup database machines storing the backup databases.
- **/.:/subsys/dce/sec** - This entry contains security specific names.
- **/.:/subsys/dce/sec/master** - This is the server entry for the master security server for this cell.

2.3.6 Summary: DCE Naming

Let us summarize the naming convention with an example of a file in a Distributed File System (DFS). The file name is *local/bin/ghostscript*, a tool used to view postscript files. We want to make this available in the shared file system of the DCE cell *itso.austin.ibm.com*.

Users within the itso.austin.ibm.com cell execute this command in one of the following ways:

- `/:/local/bin/ghostscript`
- `/.:/fs/local/bin/ghostscript`
- `/.../itso.austin.ibm.com/fs/local/bin/ghostscript`

Of course, they would normally use the first option. Users of other cells would find this file only by specifying the third command:

- `/.../itso.austin.ibm.com/fs/local/bin/ghostscript`

This is DNS naming. If the cell had been defined with X.500 syntax, the global access could only be established with the following command:

- `/.../C=US/O=IBM/OU=ITSO/CN=AUSTIN/fs/local/bin/ghostscript`

2.4 Cell Directory Service (CDS)

The CDS manages to locate names within the cell and is optimized for local access. It is a partitioned, distributed database service, and the partitions can be stored in different locations, thus allowing good scalability. The CDS can also be replicated which allows good availability of the system. A cache mechanism improves the performance by reducing the number of times it needs to be accessed.

This section introduces the components involved in the look-up of resources, the CDS database structure and how database look-ups are performed.

2.4.1 CDS Components

Like any other DCE application, it follows the client/server model. The CDS server manages a database where a CDS server adds, modifies, deletes, and retrieves data on behalf of client applications. This database is called a ***clearinghouse***. Each DCE machine runs a CDS clerk which intermediates between the client applications and the CDS server. The clerk receives a request from the DCE application to store or retrieve information and sends the request to the CDS server for processing. The clerk caches the results of look-ups so that it does not have to repeatedly go to a server for the same information. The cache is written to disk periodically so that the information can survive an application or DCE services restart. The cache is deleted upon reboot of the machine.

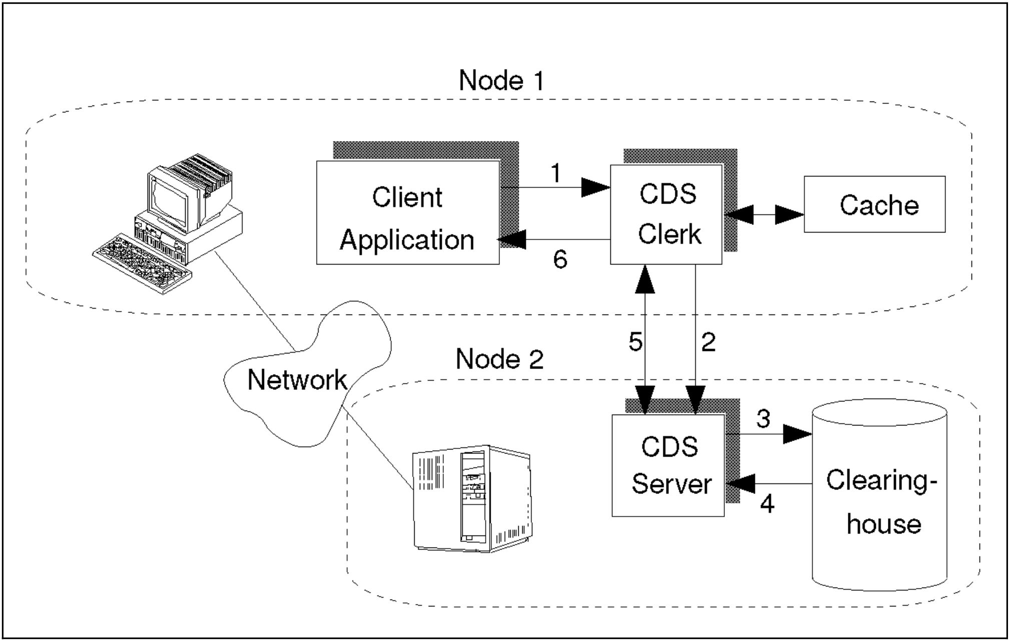


Figure 15. CDS Components Performing a CDS Look-up

Figure 15 shows the look-up process:

1. The client application on node 1 sends a look-up request to the local clerk.
2. The clerk checks its cache and, not finding the name there, contacts the server on node 2.
3. The server checks to see if the name is in its clearinghouse.

4. The name exists in the clearinghouse; so the server gets the requested information.
5. The server returns the information to the clerk on node 1.
6. The clerk caches the information and passes the requested data to the client application.

2.4.2 CDS Database Structure

The namespace is hierarchically organized and forms a tree, with containers (directories) and leaf objects. The data structure is very much like a UNIX file system. There, we have directories that build the branches of the tree. They are used to group information together. Directories can hold other directories or files. The files are the leaves of the tree and are actually the pieces that store information. Any part of the whole file tree can be a separate file system mounted to a directory. Such a file system can be imported from another machine. Furthermore, symbolic links can be used to give alias names to files and directories. With symbolic links, you can actually overlay parts of the file tree with another structure, which can become confusing and hard to track down.

The CDS structure is basically the same, but the terminology is different. See Figure 16 on page 31 for an example. A directory is called a ***replica*** because it is the unit that can be distributed and/or replicated to another clearinghouse. A directory can be duplicated. All duplicates (or replicas) of a directory contain exactly the same information and build the ***replica set*** for that directory. The replica set contains two types of replicas:

- Master replica (exactly one)
- Read-only replica (none, one, or many)

The *master replica* is the read/write instance of a specific directory in the namespace. The first instance created is automatically the master replica.

A *read-only replica* is a copy of a directory that is available only for looking up information. CDS does not create, modify or delete names in read-only replicas. It simply updates them with changes made to the master replica.

After adding read-only copies of the directory to the replica set, it is possible to select another replica to be the master. This is done through redefining the replica set with the following command:

```
# cdscp set directory <directory_name> to new epoch master /.:/ev1_ch \
readonly /.:/ev4_ch /.:/ev3_ch
```

Replicas can contain three kinds of entries:

- ***Object entries*** — The analogy to a file in the UNIX file system is a CDS leaf object or object entry. It consists of a name and attributes stored within the object. It usually contains binding information (addresses) to RPC servers, but it can also store any user-defined information. A *clearinghouse object entry* contains a list of directories contained in the respective clearinghouse as well as the address of the CDS server managing it.
- ***Soft links*** — Like a symbolic link in the UNIX file system, a soft link is a pointer that provides an alternate name for an object entry, directory or another soft link in the namespace. With soft links, it is possible to do minor restructuring of a namespace or give multiple names for an object so that different users can refer to a name that makes more sense to them.

- ***Child pointers*** — Child pointers are nothing more than subdirectories. However, instead of storing the subdirectory within the parent directory, CDS only stores a pointer to all instances of the (otherwise self-sufficient) subdirectory.

Not only can directories be replicated, they can also be distributed. This means that the master replicas of all directories can be spread over different clearinghouses, and a particular clearinghouse does not need to contain all directories from the root all the way down to a leaf object. See 2.4.3, "CDS Namespace Example" for more explanations on the CDS structure and replication.

2.4.3 CDS Namespace Example

Let us look at an example to explain the CDS terminology and replication. Figure 16 shows parts of possible cell namespace.

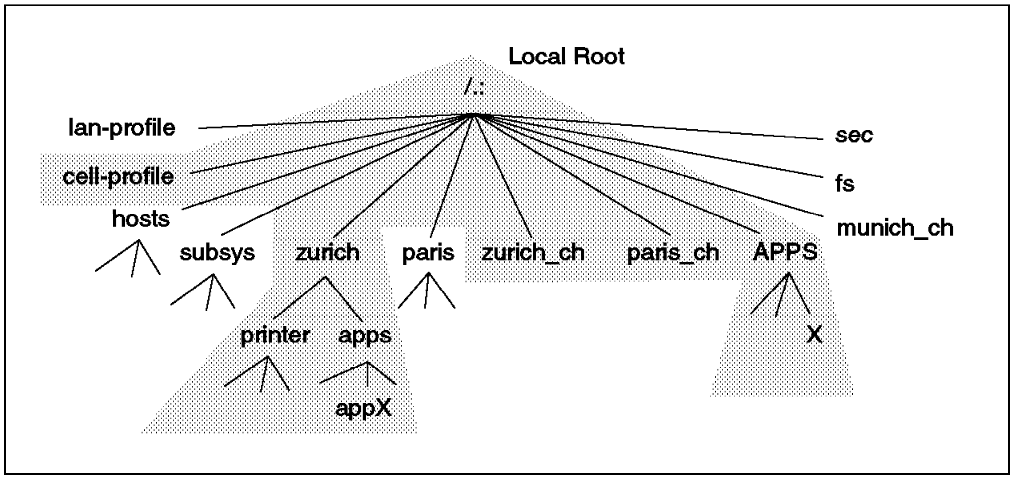


Figure 16. Cell Namespace Example

The namespace is organized in a way that groups objects together which are location dependent. Besides the usual entries, such as the lan-profile, the cell-profile, the hosts and subsys directories, the clearinghouse objects, and the sec and fs junctions, we have a directory for each major location of a fictitious company right underneath the local root level. So, for instance, objects that are mostly accessed and updated in Zurich, such as specific print servers, printers or location-specific applications, are in separate directories.

Figure 16 does not illustrate where the replicas are stored. This is pretty independent from the tree structure. The whole tree could be stored in one single clearinghouse, and it could be replicated in another clearinghouse for backup, performance and availability reasons. It is good practice to start with such a CDS namespace configuration. Later, when the cell is growing, the location-specific directories can be replicated and their master replica moved to new clearinghouses at these locations without restructuring the namespace.

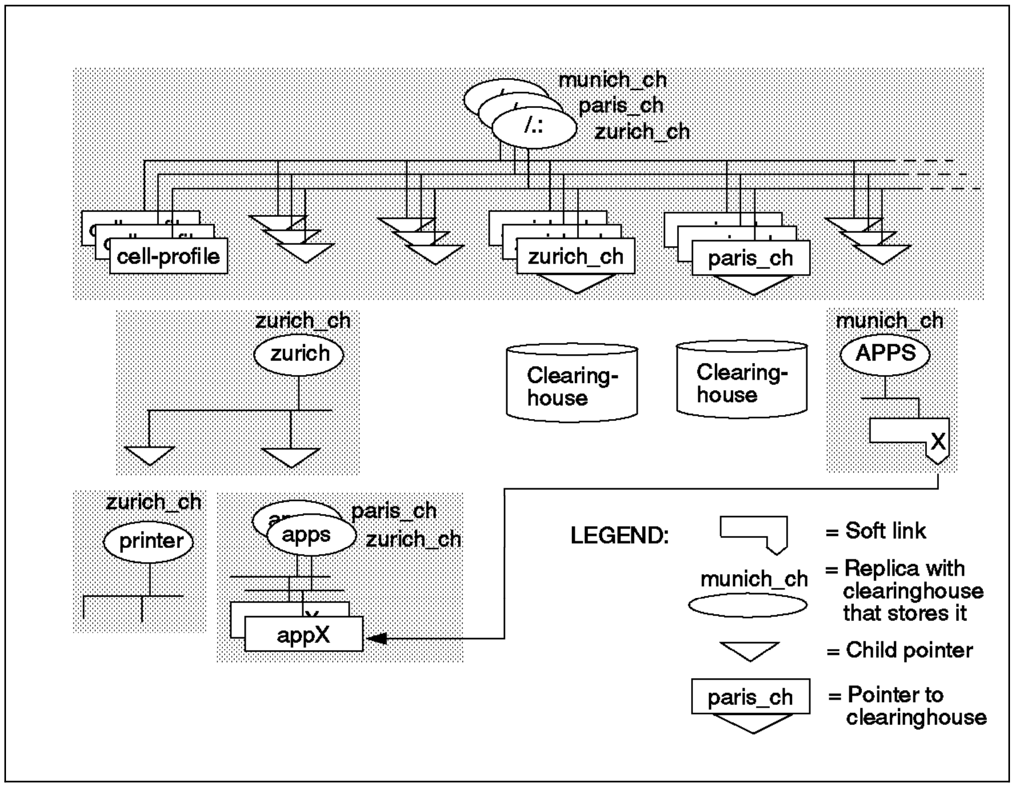


Figure 17. Cell Namespace Showing Replication and Sites

Figure 17 illustrates replication details of the shaded part of the CDS tree shown in Figure 16 on page 31. Cell root′s replica set consists of three (identical) replicas in clearinghouses in Munich, Paris and Zurich. The root directory contains the first hierarchy level of the CDS tree, such as the cell-profile, the child pointer to the zurich subdirectory, the zurich_ch and paris_ch clearinghouse entries, and a child pointer to the APPS directory.

The /.:/zurich subdirectory has only one replica and is stored in the zurich_ch clearinghouse. This replica contains two child pointers to the /.:/zurich/printer and /.:/zurich/apps subdirectories.

The /.:/zurich/apps replica set consists of two replicas, one in Zurich and one in Paris. The company′s Paris location apparently wants to be independent of the availability of the CDS server in Zurich and to be able to quickly obtain the contents of this directory. This means the clearinghouse paris_ch managed by a CDS server in Paris will contain a read-only replica of the /.:/zurich/apps directory.

The /.:/APPS directory is only in the munich_ch clearinghouse and contains a soft link. Users that want to access /.:/zurich/apps/appX can do so by specifying the alternative name /.:/APPS/X.

To further clarify Figure 17, we can list the replicas contained in each clearinghouse:

- zurich_ch

```
/.:
/.:/zurich
/.:/zurich/printer
/.:/zurich/apps
```

- paris_ch

  ```
  /.:
  /.:/zurich/apps
  ```

- munich_ch

  ```
  /.:
  /.:/APPS
  ```

2.4.4 CDS Lookup

CDS stores the names and attributes of resources in the local cell. Although users are free to define their own objects and attributes, in most cases the attributes store network address of servers that provide access to the resources. The address attribute is called CDS_Towers and can contain several addresses of compatible servers.

Knowing the resource name, a client program can look up the address in the CDS. The RPC client uses the Name Service Interface (NSI) API to get binding information from the CDS. As explained in 2.4.1, "CDS Components" on page 29, a CDS clerk performs the look-up on behalf of the client application.

The CDS clerk first checks its local cache. When the cache is empty, it might have to perform several look-ups to resolve a long pathname. For example, in Figure 17 on page 32, a client in Munich that only knows the munich_ch clearinghouse wants to access the /.:/APPS/X application. The /.:/APPS directory is in the Munich clearinghouse. But since /.:/APPS/X is a soft link with a CDS_LinkTarget attribute, this look-up resolves to /.:/zurich/apps/appX. From the Munich clearinghouse, the CDS clerk learns about the /.:/zurich directory and its location (the Zurich clearinghouse). It then accesses the zurich_ch clearinghouse. The /.:/zurich directory contains the child pointer to the apps subdirectory. From the CDS_Replicas attribute of this child pointer, it learns that the /.:/zurich/apps directory is in Zurich and Paris, and it can access the /.:/zurich/apps/appX object in one of these clearinghouses.

Each clearinghouse has a copy of the root directory, and the root directory has all clearinghouse entries. And since a clearinghouse object entry has the address of the CDS server that manages it and a list of all replicas contained in the clearinghouse, each clearinghouse knows about the locations of all replicas of the namespace.

If a clearinghouse does not contain a requested object, it provides to the caller as much information as it can, for instance other clearinghouses. Once the CDS clerk has more information, it goes straight to nearest clearinghouse that supposedly contains the required entry.

2.4.5 CDS Clerk

Before a CDS clerk is able to request a lookup to the CDS server, the CDS clerk has to learn how to locate one CDS server in the DCE environment. There are three ways for the clerk to locate them:

- Solicitation and advertisement protocol
- During a look-up
- Management command

The ***advertisement*** protocol is used by the CDS server to broadcast messages at regular intervals to advertise its existence to clerks in a LAN. The advertisement message contains information, such as the cell that the server belongs to, the server′s network address and the clearinghouses it manages. At startup, the clerks send out ***solicitation*** messages (broadcasts) to request for advertisement.

During a look-up, if the CDS server does not find the name in its clearinghouse, it gives the clerk as much data as it can about where else to search for the name. For example, if its clearinghouse contains replicas that are part of the full name being looked up, it returns data from a relevant child pointer in the replica it does have. The child pointer′s CDS_Replicas attribute contains information of clearinghouse names and binding information.

The CDS administrator can use the `dcecp` program with the `cdscache create` subcommand to create knowledge in the clerk′s cache about a server. This command is useful when the server and the clerk are separated by a wide area network (WAN) where the advertisement protocol does not work.

2.5 Security in CDS Environment

The CDS, as any other DCE service, is integrated into the security service. The CDS server only completes an operation over the clearinghouse if the user is authenticated and authorized by the Security Service. It is a two-way process where the user or the principal is first authenticated to prove who he is and then authorized to do certain operations.

CDS authorization allows you to control user access to:

- Names in the namespace, including clearinghouses, directories, object entries, soft links, and child pointers
- Execution of privileged CDS clerk and server commands

Access control is done by creating *access control lists (ACL)* that contain individual ***ACL entries*** that determine which user (principal) can use the name and what management operations they are allowed to perform on it.

CDS ACL management software, incorporated into all CDS clerks and servers, performs access checking for incoming requests. When a principal requests an operation on a CDS name or a privileged operation on a CDS clerk or server, ACL management software examines the ACL entry associated with that name or principal name and grants or denies the operation.

2.6 CDS Administration

The DCE control program `dcecp` is the tool for the CDS administration tasks in OSF DCE 1.1. The CDS control program `cdscp` of OSF DCE 1.0.x is still supported, and in fact, a few CDS management tasks supported by `cdscp` cannot be performed with `dcecp`.

Table 9 (Page 1 of 2). cdscp Commands Not Supported by dcecp

Commands	Descriptions
disable clerk	Stops the execution of the CDS clerk.
disable server	Stops the execution of the CDS server.

Table 9 (Page 2 of 2). cdscp Commands Not Supported by dcecp

Commands	Descriptions
set cdscp confidence	Sets the confidence level for cdscp calls. High confidence means that the clerk is forced to look entries up in the CDS server rather than in its cache.
set dir to new epoch	Redefines the replica set, designating a new master replica.
show cdscp confidence	Shows the confidence level in effect.
show cell	Shows information about the cell needed for entries in DNS or GDS.
show clerk	Displays counter attributes of the CDS clerk.
show server	Displays counter attributes of the CDS server.

The cdsli provides a recursive listing of directories and objects in the whole namespace.

A further CDS administration program is the ***CDS namespace browser***. The CDS namespace browser is a client application that allows you to inspect the cell′s namespace. The browser is based on the OSF/Motif graphical user interface. It allows you only to show the DCE namespace, not to change or modify it. You can start the browser with the cdsbrowser command.

In addition to the CDS control programs, other DCE interfaces allow access to, and management of, CDS names. The access control list (ACL) editing command (acl_edit) supplied with the DCE security service is used for users to control access to CDS directories and their contents.

The following sections will describe how to use the CDS user interfaces.

2.6.1 CDS Control Programs dcecp and cdscp

The CDS control programs are interactive command line interfaces with which you can manage the components of the CDS and the contents of the namespace. To start the CDS control programs, enter:

```
# dcecp
dcecp> quit
# cdscp
cdscp> quit
```

Instead of quit, you can use q or exit to leave the CDS control programs. The dcecp command contain the following elements:

```
dcecp> <object> <verb> [argument] [options]
```

where:

object	The CDS managed objects are: • directory • link • object • clearinghouse • cdscache
verb	Action to be taken
argument	Affects the result of the action specified
options	Associates the options to that object

To list all options for an object operation, you can use the following command:

```
dcecp> <object> help <verb>
```

2.6.2 Viewing the Namespace

The CDS control program, dcecp, allows you to view the namespace structure and contents.

2.6.2.1 Viewing a Cell

You can show the attributes of a cell configuration with:

```
dcecp> cell show
{secservers
 /.../itsc1.austin.ibm.com/subsys/dce/sec/master}
{cdsservers
 /.../itsc1.austin.ibm.com/hosts/ev1}
{dtsservers
/.../itsc1.austin.ibm.com/hosts/ev1}
{hosts
/.../itsc1.austin.ibm.com/hosts/ev1}
```

2.6.2.2 Viewing a Clearinghouse

To show the clearinghouse names, use the following command:

```
dcecp> clearinghouse catalog
/.../itsc1.austin.ibm.com/austin_ch
/.../itsc1.austin.ibm.com/ev1_ch
```

The dcecp> clearinghouse show /.:/ev1_ch returns the attributes of the clearinghouse /.:/ev1_ch

2.6.2.3 Viewing a Directory

The following command list the descendants (contents) of a directory /.:/ from the current cell:

```
dcecp> directory list /.:/
```

With the show command, you can see the attributes of a directory:

```
dcecp> directory show /.:/
```

2.6.2.4 Recursively Listing the Namespace

The cdsli command can selectively list parts of the namespace tree structure. The following command lists everything:

```
# cdsli -cworld
```

2.6.2.5 Viewing an Object

The following command displays all of the object entries stored in the /.:/sec directory:

```
dcecp> object show /.:/sec
```

2.6.3 Managing Clerks, Servers and Clearinghouses

CDS clerks, servers and clearinghouses are initially created and started as part of CDS clerk and server configuration. They are largely self-regulating and, apart from monitoring, require minor management intervention.

2.6.3.1 Displaying Information

Every clerk, server and clearinghouse maintains a set of attributes called counters to keep track of the operations performed since it was last started up. To list these counters, issue the following commands:

- Display clerk counters:

  ```
  cdscp> show clerk
  ```

- Display server counters:

  ```
  cdscp> show server
  ```

- Display clearinghouses counters:

  ```
  cdscp> show clearinghouse /.:/ev1_ch
  dcecp> clearinghouse show /.:/ev1_ch
  ```

Just as an example, when you display the server counters, you get the following answers:

```
cdscp> show server
                        SHOW
                      SERVER
                          AT   1995-06-01-10:06:33
               Creation Time = 1995-05-31-15:24:20.077
            Future Skew Time = 0
             Read Operations = 6409
            Write Operations = 33
            Skulks Initiated = 7
            Skulks Completed = 7
   Times Lookup Paths Broken = 0
            Crucial Replicas = 0
       Child Update Failures = 0
           Security Failures = 0
        Known Clearinghouses = /.../itsc1.austin.ibm.com/ev1_ch
```

2.6.3.2 Starting and Stopping CDS Entities

To manage clerks and servers on the local machine, you need cdscp with the following subcommands:

- To disable a clerk, enter:

  ```
  cdscp> disable clerk
  ```

- To disable a server, disable the clerk first:

  ```
  cdscp> disable clerk
  cdscp> disable server
  ```

- To restart a CDS client, you need to start the advertiser:

  ```
  # cdsadv
  ```

- To restart a CDS server, you need to start the advertiser first and then the CDS daemon:

  ```
  # cdsadv
  # cdsd
  ```

2.6.3.3 Backing up the Clearinghouse(s)

If the CDS database becomes corrupted, for example, after the system time had been set forward by mistake, you should be able to restore a consistent level of CDS rather than having to reconfigure the whole cell. To back up the CDS, do the following on all CDS servers:

1. Log into the CDS server system that contains the clearinghouse.
2. Stop the CDS server as explained above. The server writes the in-memory copy of its clearinghouse to checkpoint files.
3. Backup all files associated with a clearinghouse, for example, ev1_ch:

   ```
   # tar -cvf/dev/rmt0 /var/dce/directory/cds/*ev1_ch.* \
   /var/dce/directory/cds/cds_files /etc/dce/cds_attributes \
   /etc/dce/cds_config  /etc/dce/cds.conf
   ```

4. Reactivate the CDS server.

2.6.3.4 Restoring the Clearinghouse(s)

To restore any of the clearinghouses, perform the following steps:

1. Stop DCE.
2. Restore the database(s).
3. Restart DCE - or better reboot the system to clean out all caches.
4. Clean all clerk caches as described below.

2.6.3.5 Cleaning up the Clerk Cache

To clean up a CDS clerk′s cache, perform the following steps:

1. Log into the CDS clerk system on which you want to clean the clerk cache.
2. Stop the CDS client as explained above. If the system runs a CDS server, stop the CDS server.
3. Remove the cache files:

   ```
   # rm /opt/dcelocal/var/adm/directory/cds/cds_cache.*
   # rm /opt/dcelocal/var/adm/directory/cds/cdsclerk_*
   ```

4. Reactivate the local CDS entity.

2.6.4 Managing CDS Directories

In small networks, you can maintain all your names in the root directory and may not need to create additional directories. However, for larger networks, you should consider creating at least one additional level of directories.

2.6.4.1 Creating a Directory

CDS cell configuration creates an initial hierarchy of directories under the root directory to provide DCE components with fixed locations within the namespace where they can create and catalog their object entries.

To create a directory, you need the following permissions:

- Insert permission to the parent of the new directory.
- Write permission to the clearinghouse that will store the master replica.
- The server principal for the server system where you enter the create command must have read and insert permission to the parent directory of the new directory.

To create the /.:/subsys/PrintQ directory as *master replica*, use the following command:

```
dcecp> directory create /.:/subsys/PrintQ
```

2.6.4.2 Creating a Replica

When creating a replica, it is important to verify that the clearinghouse containing the master replica of the directory is running and reachable.

To create a read-only replica of the /.:/subsys/PrintQ directory in the /.:/austin_ch clearinghouse, use the following command:

```
dcecp> directory create /.:/subsys/PrintQ -replica -clearinghouse /.:/austin_ch
```

With the dcecp> directory show /.:/subsys/PrintQ command, you′ll see the change.

2.6.4.3 Deleting a Replica

To delete the replica, use the following command:

```
dcecp> directory delete /.:/subsys/PrintQ -replica -clearinghouse /.:/austin_ch
```

2.6.4.4 Skulking a Directory

The skulk operation is a periodic distribution of recent modifications made to the namespace. CDS skulks every directory at regular intervals according to the value assigned to the directory′s CDS_Convergence attribute. You can synchronize the replica set rather than waiting for the next scheduled skulk with the following command:

```
dcecp> directory synchronize /.:/subsys/PrintQ
```

Ensure that every replica in the directory′s replica set is reachable.

2.6.4.5 Merging or Appending a Directory

Reorganizing the namespace can be done with soft links. This is one way to do it which also worked in OSF DCE 1.0.x. However, the information is not really moved. OSF DCE 1.1 introduces new ***subtree commands*** that allow you to actually move or delete entire subtrees to another directory.

Merging a directory means the contents of a directory (object entries, child pointers and soft links) are copied over to another directory that might already contain other elements. For instance, the /.:/zurich directory in Figure 16 on page 31 could be merged into the /.:/APPS directory and then be deleted at the old place with the following commands:

1. Perform a skulk on the /.:/zurich directory to synchronize all replicas:

   ```
   dcecp> directory synchronize /.:/zurich
   ```

2. Perform the merge with the *-tree* option to perform a recursive copy including subdirectories:

   ```
   dcecp> directory merge /.:/zurich -into /.:/APPS -tree
   ```

3. Delete the /.:/zurich directory after synchronizing it:

   ```
   dcecp> directory synchronize /.:/zurich
   dcecp> directory delete /.:/zurich -tree
   ```

4. You can create a soft link from the old to the new location so that the old path is still available:

```
    dcecp> link create /.:/zurich -to /.:/APPS
```

The /.:/APPS directory then has the following contents:

```
# cdsli -world /.:/APPS
l     /.:/APPS/X
d     /.:/APPS/apps
o     /.:/APPS/apps/appX
d     /.:/APPS/printer
```

Appending a directory means copying the whole source directory (not just its contents) into a target directory. This is achieved by first creating a new directory and then merging the contents of the source directory into the new directory. In the above example, you would first create a /.:/APPS/zurich target directory and then merge the /.:/zurich directory into the /.:/APPS/zurich directory by following the above steps. This would yield the following structure:

```
# cdsli -world /.:/APPS
l     /.:/APPS/X
d     /.:/APPS/zurich/apps
o     /.:/APPS/zurich/apps/appX
d     /.:/APPS/zurich/printer
```

2.7 Platform-Specific Implementation

This section gives a summary of the platform-specific implementation differences for the CDS.

2.7.1 Directory Service on AIX Version 4

The directory service on AIX is an implementation of the OSF DCE 1.1 directory service. It contains the CDS client, the CDS server and the Global Directory Agent (GDA). The Global Directory Service (GDS) is not provided in this AIX DCE Release 2.1. However, GDS can exist in the same cell and be used for intercell communication if its provided by an another product, such as AIX DCE 1.3. Also, the hierarchical cell and the cell name aliasing features are not supported in the current AIX DCE 2.1 version. They will be supported in the next version.

2.7.2 Directory Service on AIX Version 3.2

IBM DCE Version 1.3 for AIX Version 3.2 supports all CDS components on the level of OSF DCE 1.0.3 as well as the Global Directory Service (GDS).

2.7.3 Directory Service on OS/2 Warp

The current OSF DCE 1.1 for OS/2 prototype provides the implementation of the CDS client, the CDS server and the Global Directory Agent (GDA). The Global Directory Service GDS (client and server) are not supported on OS/2.

2.7.4 Directory Service on DOS Windows

DCE Directory Service on DOS Windows is only a CDS client service implementation. The Global Directory Service GDS (client and server) is not supported on DOS Windows.

Chapter 3. Security Service

Distributed computing encourages a free flow of data between nodes, thus expanding the capabilities of interconnectivity and interoperability. Security breaches might come from any component of the distributed system. The Security Service is a strong building block of the DCE core services that provides secure authentication, authorization and auditing mechanisms for users and distributed client/server applications.

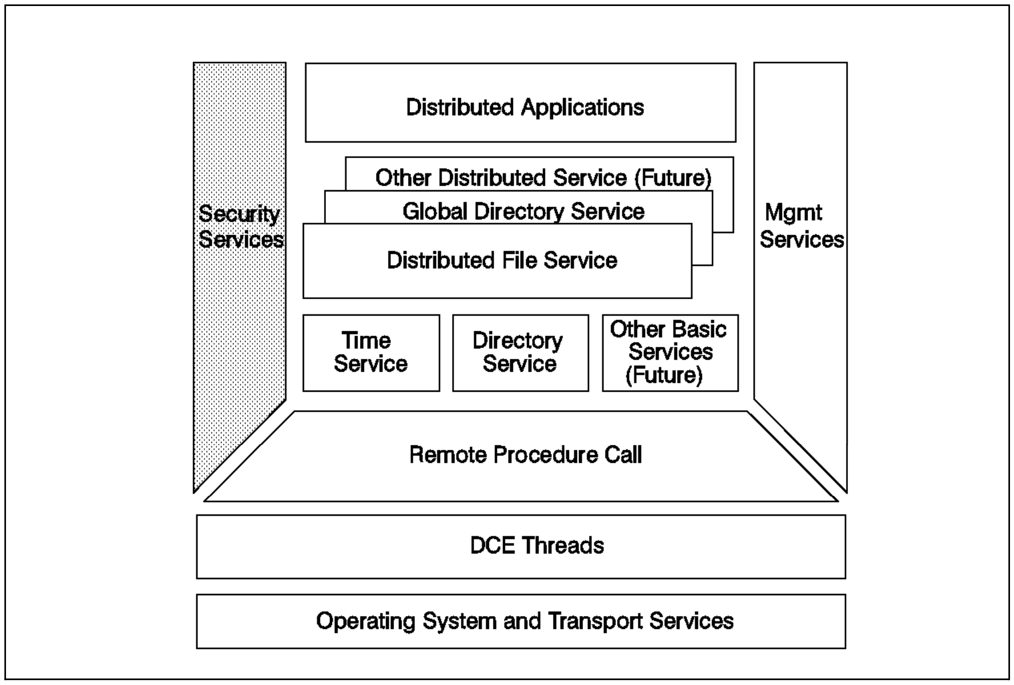


Figure 18. DCE Architecture: Security

Security is one of the main reasons why customers are interested in DCE. Developers can use the DCE Security Service to make their distributed client/server applications or products secure. They do not necessarily need to use the DCE RPC API. The GSS-API allows interaction with the DCE Security Service without using any other DCE components. In their Open Blueprint strategy paper, IBM announced that it would integrate the DCE Security Service into other products to provide them a higher level of security.

Administrators must understand all the concepts and components of the Security Service. As other products, such as DB2 or LAN Server, imbed a DCE security layer, the community of administrators with DCE knowledge will grow above the pure DCE administrator community.

Users will not notice much of DCE security, if they do not want to. In some systems, they have to perform a separate login to the DCE to obtain their network credentials, but this DCE login will be integrated more and more into the operating system login, resulting in a single login. For instance, AIX Version 4 provides an authentication method that contacts the DCE registry to obtain login credentials for the local system. Users might have to understand Access Control Lists (ACLs) to protect their own objects. However, administrators can set up default ACLs that are inherited when new objects are created so that users do not even have to deal with ACLs.

In this section, we will explain the concepts and the components of the DCE Security Service.

3.1 Open Systems and Security

In a picture of modern computing, hundreds of heterogeneous systems interoperate with each other and are connected via local and wide area networks. For a variety of procedural, physical, automated, and historical reasons, each of the various computers and networks carry with them implicit or explicit levels of trust or threats.

Networks could create larger holes and greater threats than a single system. Network security is becoming a big concern for several companies because anyone can have access to the various systems′ resources. Nowadays networks are considered part of the system. However, trusted networks have been studied much less than trusted computer systems. One of the major reasons is the complexity of the distributed environment. Security standards have been developed for network security, but very few systems, so far, have been evaluated for conformance.

A security threat is often thought of as hackers from outside trying to get access to critical data. However, it will take the hacker some time to realize where the valuable data is located. As a matter of fact, most of the security problems are generated internally from employees within the company or former employees who have some inside knowledge. They cannot be blindly trusted. It is important that your management understands security and sets a policy. If in the company security is considered too expensive, the management is not able to estimate the value of the information asset it is managing.

Security refers to protection against unwanted disclosure and modification of data and files in a system. It also refers to the safeguarding of systems themselves. Education in security is important. Users, developers, system administrators, and customers should know how to buy and use systems that implement security.

The rest of this section will discuss the main security requirements, security policies and security standards.

3.1.1 Security Requirements

Open Systems stands for portability, scalability and interoperability, which suggests some degree of anarchy and disorder. However, a system can be *open* and *secure* at the same time. Security threats can be:

- Eavesdropping: Data can be read as it flows over the network.
- Masquerading: A system can pretend to be another system and thus gain unauthorized access to resources.
- Modification: Data can be modified as it flows over the network.
- Denial of service: Service can be denied from an unauthorized source.

These are just a few of the problems that can arise. A secure environment has to fulfill the following requirements:

- Confidentiality: The information is disclosed only to authorized users. Passive wire-tapping can be a form of security threat that breaks confidentiality.
- Integrity: The information is modified only by authorized users. Active wire-tapping is a form of security threats that breaks integrity.
- Availability: The use of systems, applications and services cannot be maliciously denied to authorized users.
- Accountability: Users are accountable for actions relevant to their security.

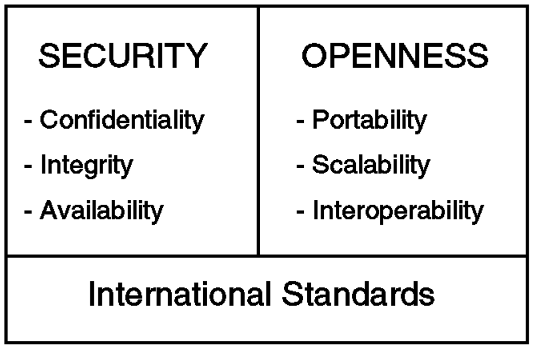


Figure 19. Security Requirements

Figure 19 illustrates the prevailing requirements of Open Systems and security. Both arenas are based on international standards.

3.1.2 Security Policies

The particular system security needs will vary from organization to organization and, within them, from application to application. As a result, organizations must both understand their applications and think through the relevant choices to achieve the appropriate level of security. Then, they must set the appropriate policy.

A ***discretionary policy*** requires that single individuals protect their own assets. However, a user with super-user authority can bypass any level of protection that has been set. A ***mandatory policy*** is set at the organizational level and is not controlled by the users.

All countermeasures should be taken in order to handle threats that might exploit a vulnerability of the system. A *recovery policy* must be in place, independent from the policy the organization has chosen, if the system has been compromised. A *threat analysis* should be conducted in your organization to evaluate the resources that are at risk.

Incident reporting and tracking is also important. If a hole is identified in a distributed or widely available software package, it would be better to be prudent and not to publicize the problem. Report the bug quietly to the vendors and to the security crisis centers, such as the CERT (Computer Energy Response Team) that monitors computer security or the CIAC (Computer Incident Advisory Capability).

3.1.3 Security Standards

Characterizing a computer system as being secure presupposes some criteria, explicit or implicit, against which the system in question is measured or evaluated.

The evaluation criteria reflect two independent aspects: *functionality* and *assurance*. The first one refers to the facilities by which security services are provided to users, such as:

- Identification
- Authentication
- Access Control
- Auditing

A product rating intended to describe security assurance expresses an evaluator′s degree of confidence in the effectiveness of the implementation of security functionality. Assurance criteria, which are not user-visible, are the following:

- Integrity
- System Management
- Object Reuse

In 1983, the National Computer Security Center (NCSC) published the Trusted Computer System Evaluation Criteria (TCSEC), best known as the Orange Book. Later in 1987, the Trusted Network Interpretation (TNI) and the series of Rainbow books, including the Green Book for Password Management, were published. It considers the network and its various interconnected components as special occurrences of a trusted system.

In the Orange Book publications, NCSC established ratings that span four hierarchical divisions: D, C, B, and A in ascending order. For each division, one or more classes are defined for a total of seven classes. The ratings reflect increasing provisions of security:

- Division D — No Guaranteed Security

  Class D1: No Security

- Division C — Discretionary Access Control
  - Class C1: Discretionary
  - Class C2: Controlled Access Protection
- Division B — Mandatory Access Control
  - Class B1: Labeled Security Protection
  - Class B2: Structured Protection
  - Class B3: Security Domains
- Division A — Verified Model
  - Class A1: Verified Design

Each class bundles both security functionalities and assurance. When we talk about security classes, we will always refer to the Orange Book classes (D1, C1, C2, B1, B2, B3 and A1). In 1985, the U.S government directed that all computers used in government agencies and organizations have a C2 rating by 1992 and a B1 rating by 2001. DCE implements C2 security.

Several other countries, such as Canada, Sweden and Australia, developed their own evaluation criteria. France, Germany, the Netherlands, and the U.K. joined to write a security evaluation criteria document and named it the ITSEC (Information Technology Security Evaluation Criteria).

Organizations, such as POSIX, X/Open and ISO/OSI, are focusing on some functional aspects of security. The International Standards Organization (ISO), with the SC27 group, focuses on security techniques for encryption and authentication, but X.400 and X.500 also consider security. X/Open has published in the X/PG4 an API for auditing. POSIX, with the group 1003.6, has published in draft form the model for the access control list (ACL). DCE implements a superset of POSIX ACL Draft.

3.2 DCE Security Service Components and Facilities

The DCE Security component comprises three services running on the security server and several other facilities. Most of the DCE security is related to the concept of a principal. A ***principal*** is an entity that can be securely identified and can engage in a trusted communication. A principal usually represents a user, a network service, a particular host, or cell. Each principal is uniquely named and identified by its principal UUID. A record for each principal containing the name, the private keys and the expiration date is kept in the registry database on a highly secure system.

The three services are:

- ***Registry Service (RS)*** — A replicated service which maintains the cell′s security database. This database contains entries for accounts, principals, groups, organizations, and administrative policies.
- ***Authentication Service (AS)*** — Used to verify the identity of principals. It contains a Ticket-Granting Service (TGS) which grants tickets to these principals and services so that they can engage in a secure communication.
- ***Privilege Service (PS)*** — Certifies a principal′s credentials that are going to be forwarded in a secure way to DCE servers. The credentials (see EPACs below) allow the target server to check the principal′s access rights to resources.

These services are implemented in the ***security server daemon*** (secd). The DCE services are considered the DCE′s Trusted Computing Base (TCB). If security is compromised for any of those services, the security of all DCE is compromised. It is important that the security daemon runs on a secure, and highly available, computer under the control of a dedicated system administrator or, better yet, an Information System Security Officer (ISSO). The registry database is only as secure as the security provided by the machine on which it resides.

On each client machine, there is a ***Security Validation Service*** integrated into the DCE daemon process. In OSF DCE 1.0.x, this used to be the `sec_clientd` process (AIX) or `sclientd` (OS/2). It has the following duties:

- Verifying that the security server is authentic
- Managing the machine principal (see also 3.2.3.6, “Keys and Key Management” on page 57)
- Certifying login contexts (see also 3.2.3.1, “DCE Login Facility” on page 52)

Client applications actually use the Security Service facilities and services via a Security Service API or the Generic Security Services API (GSS-API). The facilities serving these APIs are:

- ***Login Facility (LF)*** — Initializes a user′s DCE security environment (login context) and provides them with their security credentials.
- ***Extended Registry Attribute (ERA)*** — Extends the standard set of registry database attributes (which cannot be changed) and allows for user-defined attributes.
- ***Extended Privilege Attribute Certificate (EPAC)*** — Basically a certified list of the principal name, groups of which the principal is a member, and the ERAs for an authenticated principal. A client must present its EPAC to a server when performing authenticated RPC. The server uses the EPAC to examine the client′s access rights. Other information in the EPAC allows clients and servers to invoke secure operations through one or more intermediate servers (delegation).
- ***Access Control List (ACL) Facility*** — An ACL is a list of principals or groups and their access permissions. ACLs are assigned to any type of resource that DCE servers manage. The ACL facility provides a generalized means of checking a principal′s access request against the ACLs on the requested resource.
- ***Key Management Facility*** — Enables non-interactive principals (such as application servers) to manage their secret keys.
- ***ID Map Facility*** — Allows intercell communication, mapping local cell principal names to global cell principal names and vice versa.
- ***Password Management Facility*** — Enables principal passwords to be generated and to be submitted to strength checks beyond those defined in DCE standard policy.
- ***Audit Service*** — Detects and reports events that are relevant to the management of a secure environment. Events are written in a log file, called an *audit trail file*. The application programmers need to use an audit API to build auditing of relevant operations into their applications.

All of the Security Service′s events, such as creating users, logging in and giving tickets, can be recorded in the audit trail file. The Security Service is enabled to use its Audit Service.

The following sections will explain these components in more detail.

3.2.1 The Security Registry

The registry server manages the registry database where the security-relevant information of a DCE cell is stored. The registry database can be replicated as a whole. There is a master site which can updated and an arbitrary number of replica or slave sites which are read-only.

The `sec_create_db` command on AIX, or `screatdb` on OS/2, is used to create the master database. These commands are implicitly called by the AIX and OS/2 provided administration tools to configure a DCE cell. See Chapter 6, “Installation and Configuration of DCE” on page 103 for more information about these specific tools. Then the security daemon is started and initial users, groups and accounts are entered.

The administrator can create several security replica servers to balance the load on the master security server and to preserve the cell in case the master becomes disabled. The sites where the security database will be replicated must be as secure as the site where the master copy of the security database is stored.

The registry database contains the following information, also called ***registry objects***:

- ***Policies*** — The *standard policy* regulates such things as account and password lifetime and format. It can be set for the registry and for specific organizations.

  The *authentication policy* regulates ticket lifetimes and can be set for the whole registry, an individual organization or an individual account.

- ***Properties*** — They define such things as the default certificate (ticket) lifetimes and the range of UNIX, group and organization IDs. This is done only for the registry as a whole.
- ***Principals*** — They are either interactive users of the system or non-interactive servers, machines and cells. Principals can be associated with access permissions.
- ***Groups*** — They are collections of principals identified by a group name. Groups can be associated with access permissions.
- ***Organizations*** — They are collections of principals identified by an organization name. Organizations do not have any access permissions; they define policies associated with the principals.
- ***Accounts*** — An account is a definition of a potential network identity for a principal. The attributes that define an account are a unique combination of principal, group and organization, also known as ***PGO***. The account contains other attributes, such as the password, home directory, whether it can be logged into it, or whether it can be a server.
- ***xattrschema*** — Such an object contains the definition of an extended registry attribute (ERA) created with the ERA facility. This is described in 3.2.2, "Extended Registry Attributes (ERA)" on page 49.
- ***replist*** — Used to manage the replicas of the registry database.

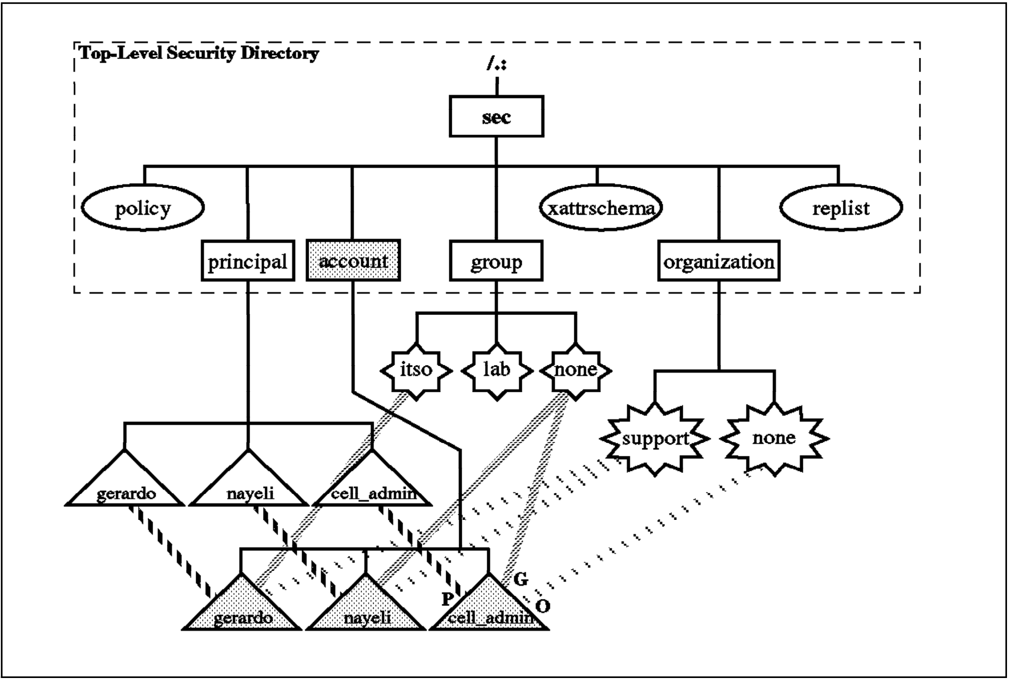


Figure 20. Registry Structure and Accounts with PGO Assignments

Figure 20 illustrates the structure of the registry. Each account is defined with a unique PGO combination. Groups and organizations can have several members. A principal is usually identified with its security namespace entry, such as, /.:/gerardo. However, commands that work on objects of different namespaces (such as acl_edit) need the full name in the CDS namespace, which is /.:/sec/principal/gerardo. See also 3.7.2, "DCE Security and Naming" on page 72 for a further discussion of names.

To create principals, accounts, groups, and organizations, the administrator uses the dcecp program (OSF DCE 1.1). The rgy_edit command, used to manage the registry in OSF DCE 1.0.x, is kept for compatibility reasons.

The registry database contains UNIX user IDs and group IDs. They are used for compatibility with UNIX programs. UNIX commands determine file ownership or access rights based on these IDs. If there is a mismatch between a principal's UNIX ID in the registry and its local definition in a UNIX machine, file ownership of DFS files can become quite confusing. It is therefore important that all UNIX machines in a cell use consistent IDs for one specific principal.

Before we can create an account, we must define a principal, and maybe a group and an organization. We must then explicitly add the principal to a group and to an organization, which was implicitly done with rgy_edit. This is illustrated in the following example:

```
dcecp> principal create nayeli
dcecp> group add users -member nayeli
dcecp> organization add ibm -member nayeli
dcecp> account create nayeli -password secret -mypwd dce \
 -group users -organization ibm
```

The use of groups and organizations simplifies the security management for the cell administrator. Using DCE's groups, we can have the same benefits as when we use groups in the UNIX environment. We can give ACL permissions to a group, and all principals in that group will have the same access permissions to the object. Administrators can use organizations to apply global security policies to several principals (the members of the organization) at once.

The registry database is maintained in the virtual memory of the security servers. All updates are made to the in-memory copy of the master and then immediately sent to the slaves. The registry is checkpointed out to the disk every two hours. This is the default value, but the ***checkpoint interval*** can be changed by restarting the secd and specifying a parameter to the secd command as follows:

```
# secd -cpi 300
```

This would force the secd to create a checkpoint every five minutes. Remember to be user *root* when you restart secd. All the changes made to the database between checkpoints are written to an update_log file. In case of a crash, the last checkpoint is recovered, and the log helps to restore the registry database.

Applications communicate with the registry server via authenticated RPC. This is documented in 10.4, "RPC and Security" on page 191.

3.2.2 Extended Registry Attributes (ERA)

Initially designed for UNIX systems, the registry contains a user ID and a group ID for each account. When accessing services on UNIX systems, DCE principals assume these IDs which eventually determine their local access permissions. Recognizing that not all the machines in a distributed environment are running some variant of the UNIX operating system, a mechanism for storing user and group attributes for arbitrary operating systems has been added in DCE 1.1: the Extended Registry Attribute (ERA).

The registry continues to use the standard set of attributes, a fixed schema, for its objects as in older versions of the registry. For example, the account still has the UID, GID, the home directory, and so on. The ERAs introduce a dynamic schema that extend the fixed schema. New attributes can be defined as *schema entries* or *attribute types*. You can then, optionally, add *instances* of these defined attribute types to registry objects as extended attributes.

So, there are basically two *administration* tasks required before ERAs can be used: managing ERA definitions and assigning ERAs to registry objects.

3.2.2.1 Managing ERA Definitions

The extensions to the registry schema are maintained in the ***extended attribute schema (xattrschema)*** object of the registry which is identified by the name *xattrschema* under the security junction point (/.:/sec) in the CDS namespace.

The /.:/sec/xattrschema object provides a catalog (directory) for all extended attributes known to the system. This catalog may be dynamically updated to create, modify or destroy schema entries (attribute definitions).

A ***schema entry*** defines the semantics (format and usage) of an attribute type and is identified by a name and a UUID. The characteristics of an attribute definition are:

- ***-encoding*** (required) — Defines the format, such as printstring, integer and UUID.

- ***-aclmgr*** (required) — The registry has a different ACL Manager for each of its objects. See 3.2.1, “The Security Registry” on page 46 for a list and description of the registry objects. By specifying one of these names for the ACL Manager characteristic, you define to which type of object the attribute can be added and which ACL Manager handles the permission check.

  More than one ACL Manager can be listed with this flag to build an ACL Manager set. It also defines the permission bits needed to query, update, test, and delete attribute instances assigned to particular registry objects.

- ***-multivalued*** — If this boolean type is set to *yes*, one single registry object can have several attributes of the same type.

- ***-unique*** — If this boolean type is set to *yes*, no two instances of this attribute can have the same value.

- ***-reserved*** — If this boolean type is set to *yes*, an instance of this attribute cannot be deleted, before it is changed to *no* (non-reserved).

- ***-trigtype*** — This option allows you to associate the access of an attribute with the automatic execution of another DCE RPC server. This may be useful, for instance, if the value has to be verified with an external database or is stored there. A trigger binding must be specified that contains a binding handle to a remote DCE server.

The following example is a first step towards login integration of DCE users into MVS. If we want to associate MVS user names with DCE principals, we can create an attribute schema named MVSname as follows:

```
dcecp> xattrschema create /.:/sec/xattrschema/MVSname -encoding
printstring -aclmgr {principal r c r D}
```

The MVSname attribute, as specified above, will accept strings and can only be assigned to principals. Access control to the MVSname attribute assigned to a particular principal is managed by the ACL of that principal (and the principal ACL Manager). So, in order to query, update, test, or delete the attribute of a principal, the caller needs to have *r, c, r,* or *D* permission, respectively, on that principal object.

To see all defined ERAs and to check the definition of one them, enter the following commands:

```
dcecp> xattrschema cat /.:/sec/xattrschema
/.../itso7.austin.ibm.com/sec/xattrschema/pre_auth_req
/.../itso7.austin.ibm.com/sec/xattrschema/pwd_val_type
/.../itso7.austin.ibm.com/sec/xattrschema/pwd_mgmt_binding
/.../itso7.austin.ibm.com/sec/xattrschema/X500_DN
/.../itso7.austin.ibm.com/sec/xattrschema/X500_DSA_Admin
/.../itso7.austin.ibm.com/sec/xattrschema/disable_time_interval
/.../itso7.austin.ibm.com/sec/xattrschema/max_invalid_attempts
/.../itso7.austin.ibm.com/sec/xattrschema/passwd_override
/.../itso7.austin.ibm.com/sec/xattrschema/test_any
/.../itso7.austin.ibm.com/sec/xattrschema/test_void
/.../itso7.austin.ibm.com/sec/xattrschema/test_printstring
/.../itso7.austin.ibm.com/sec/xattrschema/test_printstring_array
/.../itso7.austin.ibm.com/sec/xattrschema/test_integer
/.../itso7.austin.ibm.com/sec/xattrschema/test_bytes
/.../itso7.austin.ibm.com/sec/xattrschema/test_confidential_bytes
/.../itso7.austin.ibm.com/sec/xattrschema/test_i18n_data
/.../itso7.austin.ibm.com/sec/xattrschema/test_uuid
/.../itso7.austin.ibm.com/sec/xattrschema/test_attr_set
```

```
/.../itso7.austin.ibm.com/sec/xattrschema/test_binding
/.../itso7.austin.ibm.com/sec/xattrschema/MVSname
dcecp>
dcecp> xattrschema show /.:/sec/xattrschema/MVSname
{aclmgr {principal {{query r} {update c} {test r} {delete D}}}}
{annotation {}}
{applydefs no}
{encoding printstring}
{intercell reject}
{multivalued yes}
{reserved no}
{scope {}}
{trigbind {}}
{trigtype none}
{unique no}
{uuid a609d8c0-e484-11ce-87ea-10005aa86e2d}
```

DCE ACLs also control the access to the xattrschema registry object. If you want to manipulate an attribute schema, for instance, create a new attribute type definition, you need the appropriate permissions on the ACLs of the /.:/sec/xattrschema object.

3.2.2.2 Assigning ERAs to Registry Objects

Once the ERAs are defined, *ERA instances* can be assigned to, modified on or deleted from *registry object instances*. For example, we can add the above created MVSname attribute to the DCE principal nayeli:

```
dcecp> principal modify nayeli -add {MVSname a948r18}
dcecp> principal show nayeli -xattr
{MVSname a948r18}
```

The second command shows the attribute. An instance of the ERA object /.:/sec/xattrschema/MVSname is attached to the prinipal object /.:/principal/nayeli. If anybody wanted to modify nayeli′s MVSname attribute, they would need to have control (*c*) permission in the ACL of /.:/principal/nayeli.

3.2.2.3 Attribute Sets

An attribute set is a collection of attribute types that can be identified together for display or retrieval purposes. In other words, if you display an attribute that is a set, it actually displays all attributes that are members of this set.

An attribute set is defined as a multivalued attribute of encoding type *attrset*. It is handled like an attribute and can be assigned to an object. The value that is given to a set attribute when it is added to a registry object is a list of UUIDs of attributes that are to be the members of the set.

3.2.2.4 ERA API

The DCE′s Extended Registry Attribute API (consisting of sec_attr() calls) provides facilities for extending the registry database by creating, maintaining and viewing attribute types and instances and by providing information to, and receiving it from, outside attribute servers (or attribute triggers). It is the preferred API for security schema and attribute manipulations.

The DCE Attribute Interface (consisting of dce_attr_sch() calls) is provided for schema and attribute manipulation of data repositories other than the registry. It is limited to creating schema entries (attribute types) and does not provide calls to create and manipulate attribute instances or to access trigger servers.

3.2.3 Authentication Components and Procedures

The DCE Security Service allows a principal to verify if the other principals are who they say they are (authentication) and if they have the right to do what they want to do (authorization). DCE makes use of the Kerberos authentication service from the Massachusetts Institute of Technology′s Project Athena. Kerberos uses a sophisticated mechanism that meets the requirements of open computer networks.

This section describes the authentication procedure as well as features and services that enable a secure authentication.

3.2.3.1 DCE Login Facility

The DCE login facility allows users to participate in DCE. It has a user interface (the `dce_login` command) and an API. Interactive principals use the `dce_login` command (AIX) or the the `dcelogin` command (OS/2) to communicate with the security server and establish a login context, which means a principal′s identity is validated, and initial tickets are provided.

Application servers (and clients) can use the `sec_login...()` API to perform the equivalent of an interactive user login. In this way, they can establish their own login context rather than running under the identity of the principal that started them.

The syntax of the `dce_login` command is:

```
dce_login principal_name password [-c]
```

The user/principal is asked for the name and password.

```
# dce_login
Enter Principal Name:
Enter Password: <not-echoed>
```

If the authentication ends correctly, the Security Service returns the DCE credentials which consist of initial tickets, the Ticket Granting Ticket and Privilege Ticket Granting Ticket (TGT, PTGT), and certified user attributes (EPAC). The explanation for these terms will be the subject of the following sections. The credentials will be used to authenticate the user to distributed services that are accessed during the user′s session, such as DFS.

The `-c` option causes the principal′s identity to be certified. Without this option, the principal identity is only validated. The difference between the simple login validation and certification is strictly a client issue.

Validation means that you have obtained credentials which you can use over the network. This demonstrates that you share authentication information with a cell′s Key Distribution Center or authentication server.

Certification means that you have done the *validation* stage and have successfully authenticated to a service on your machine. By obtaining a valid service ticket for your machine from the Security Service, your machine (principal) can be sure that the user authentication was correct and the Security Service itself is authentic. If the certification is not done, the user could have contacted a spoofing security server to obtain a login to the local machine.

The certification is particularly useful if the operating system login and the DCE login are integrated. In order to use other DCE servers, certification is not needed because a valid ticket can only be obtained if all involved principals (including the Security Service) are authentic. If certification is requested, an entry is created in the local registry that allows a user to log in to the system when the Security Service is unavailable. However, to execute `dce_login -c`, the user needs root authority.

The `dce_login` facility, among other things, initializes the `KRB5CCNAME` environment variable. It contains the name of a file which contains the various tickets that the principal obtains during the lifetime of its initial master ticket, the Ticket Granting Ticket or TGT. The credential cache file is in the form of dcecred_xxxxxxxx on AIX, where the xxxxxxxx stands for a randomly assigned number:

```
# echo $KRB5CCNAME
FILE:/opt/dcelocal/var/security/creds/dcecred_416bb00c
#
```

On OS/2, the whole eight-character file name is randomly assigned. For example:

```
[C:\:] set KRB5CCNAME
KRB5CCNAME=FILE:C:\OPT\DCELOCAL\var\security\creds\GTUPRHJP
[C:\:]
```

/opt/dcelocal/var/security/creds/dcecred_ffffffff represents the machine principal′s credential cache file on AIX. On OS/2, the file name is MACHCTXT for the machine principal. If you want to clean up expired credential caches, you can use the `rmxcred` command. This command also allows you to selectively remove credential files for specific principals. It is recommended that you include this command in the root user′s *crontab* entry on AIX and run it regularly.

3.2.3.2 Secret-Key Authentication Steps

The authentication process between principals is accomplished through the exchange of secret messages that prove their identities to each other.

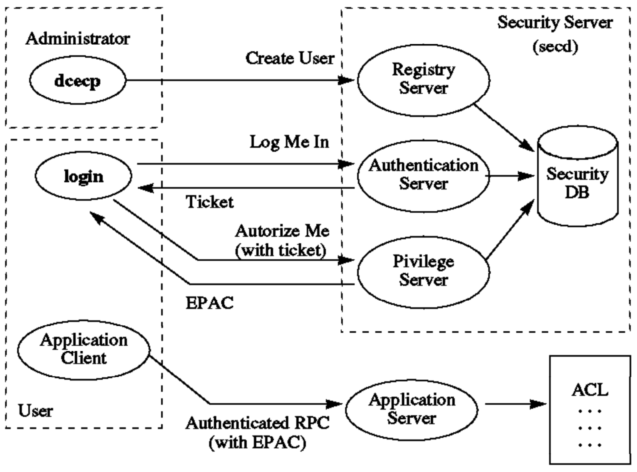


Figure 21. Authentication Process.

Figure 21 is a simplified illustration of the secret key authentication process that includes the following steps:

1. Create User

   The administrator uses the `rgy_edit` command (DCE 1.0.x), or the `dcecp` command in OSF DCE 1.1, to administer user accounts in the Registry Service.

2. Log Me In

   OSF DCE 1.1 introduces preauthentication. This prevents a security server from responding to unidentified clients so that they cannot guess user IDs and try logins anymore. See 3.2.3.3, "Preauthentication Protocols" on page 55 for more details.

   If the security client is at OSF DCE 1.1 level, the Login Facility asks for the user password at this point. If the principal initiating the login is a client or server application acquiring its own login context, the secret key is obtained from the local keytab file. If the security client is at OSF DCE 1.0.x level, the password is prompted from the user only *after* it receives the TGT.

   Then a request for a ***Ticket Granting Ticket (TGT)*** is sent to the Authentication Service (AS).

3. Ticket

   The AS performs a preauthentication if it is at OSF DCE 1.1 level. If preauthentication is successful or not done at all (OSF DCE 1.0.x), the security server sends the (TGT) and a conversation key in an envelope encrypted with the user′s secret key. The TGT itself is encrypted with the authentication server′s own key; the client cannot decrypt it.

   In OSF DCE 1.1, the password is already verified at this point, and the key can be used to decrypt the message. In OSF DCE 1.0.x, the security client can only decrypt the message if the user enters the right password or if the key stored in the keytab file is correct. It cannot discover an invalid password earlier than this.

   The security client then needs a ticket to the Privilege Service (PS). It uses the conversation key to encrypt a request to the AS, which creates a ticket that contains the principal UUID and a new conversation key and is encrypted with the PS′s secret key. Together with the new conversation key, this ticket is put into an envelope, encrypted with the first conversation key and transmitted to the client.

4. Authorize Me

   The client can decrypt the envelope, learn the new conversation key for the PS and send the ticket to the PS. Since the client does not know the PS′s secret key, it cannot decrypt or manipulate this ticket.

5. EPAC

   The PS uses its own secret key to decrypt the ticket hereby learning the principal UUID and the new conversation key. It then puts together the user′s authorization credentials. In OSF DCE 1.0.x, this was called the Privilege Attribute Certificate (PAC) and contains the principal UUID and groups of which the user is a member. In OSF DCE 1.1, the PAC is extended by the Extended Registry Attributes (ERAs) and is called Extended PAC (EPAC).

   The EPAC is sealed with a checksum encrypted in the AS′s secret key. Together with a third conversation key, the EPAC and the seal are incorporated into a ***Privilege-Ticket Granting Ticket (PTGT)***. The PTGT (except for the EPAC) is encrypted with the AS′s secret key and transmitted to the client together with the third conversation key, which is encrypted with the second conversation key.

6. Authenticated RPC

   The client can decrypt the message and learns the third conversation key that it will need to use in further conversations with the AS. It can read the EPAC, but cannot manipulate it to its own liking.

   If the client wants to call an application server, it sends a request for a service ticket to the AS together with the PTGT. The AS prepares a ticket by re-encrypting the EPAC with the application server′s secret key. This ticket is sent to the application server with the first RPC request. More on this can be found in 10.4.4, “Key Management and Secret Key Authentication” on page 193.

3.2.3.3 Preauthentication Protocols

The authentication protocol used on OSF DCE 1.0.x has a problem in that the security server always responds to the client login request without verifying that the user knows the password or where the request comes from.

OSF DCE 1.1 extends the previous authentication process. It introduces three protocols for preauthentication. Only if the preauthentication is successful does the Security Service send out the TGT as described above in 3.2.3.2, “Secret-Key Authentication Steps” on page 53. Each principal can be assigned an Extended Registry Attribute (ERA) that defines the lowest level of preauthentication that the principal can accept. If this ***pre_auth_req ERA*** is undefined or zero, then the principal has no particular preference for a protocol and can be authenticated using any protocol. The three protocols are:

- ***No preauthentication*** — This protocol is always used when an OSF DCE 1.0.x client is initiating the authentication procedure for a principal. If the principal′s ERA in the registry requires a higher protection level (pre_auth_req>=1), the login request is rejected. In other words, a principal′s pre_auth_req ERA must be undefined (or zero) to allow it to login from an OSF DCE 1.0.x client.

  OSF DCE 1.1 clients do not use this protocol. OSF DCE 1.0.x security servers always use this protocol and ignore requests of DCE 1.1 clients for higher level protocols.

- ***Timestamps protocol*** — If DCE 1.1 clients are not able to construct the *third-party* protocol, for example, because the machine′s session key is not available, they create a *timestamps* protocol login request.

  The client sends a timestamp encrypted with the principal′s secret key and the principal name. The Security Service decrypts the timestamp with the principal′s secret key stored in the registry. If the timestamp is within five minutes of the correct time, the Security Service considers this evidence that the client side principal is authentic.

  If the principal′s ERA requires a higher level of security (pre_auth_req=2), then the login request fails. If the pre_auth_req ERA of the principal is set to one, then the client *must* request at least timestamp protocol. If the pre_auth_req ERA of the principal is undefined or zero, this protocol also works, because then any protocol is accepted by the Authentication Service.

  This method prevents attackers from masquerading as a security client, but is still vulnerable to processes that monitor the network.

- ***Third-party protocol*** — This is the default protocol for DCE 1.1 clients. Basically, the procedure is the same as with the timestamp protocol, but the information (principal UUID, principal-secrect-key-encrypted timestamp) is additionally encrypted with several other keys based on the machine′s session key with the Authentication Service.

  If the pre_auth_req ERA of the principal is set to two, the client *must* request the third-party protocol; otherwise the request fails. The Authentication Service also accepts client requests using this protocol, if the pre_auth_req ERA of the principal is undefined, zero or one.

All clients, except cell_admin, should be defined with `pre_auth_req=2` once all machines are on OSF DCE 1.1 level so that only this protocol can be used. This provides the most protection against attacks. Administrator principals should be able to log in before the session key is available; so they should be defined with `pre_auth_req=1`.

If preauthentication succeeds, the Security Service sends the TGT and the normal secret-key authentication process goes on as described in 3.2.3.2, “Secret-Key Authentication Steps” on page 53.

For example, to request that a principal always authenticates using the third party protocol, we can modify the principal as follows:

```
dcecp> principal modify nayeli -add {pre_auth_req 2}
```

3.2.3.4 Invalid Login Management

In OSF DCE 1.0.x, there was no way for the Security Service to detect invalid login attempts. The password validation was done on the client side. With preauthentication and `pre_auth_req=2` in OSF DCE 1.1, the security server can detect and track invalid login attempts.

Two ERAs should be defined for each principal to determine the number of invalid login attempts permitted before marking an account as disabled (***max_invalid_attempts ERA***) and for how long (in minutes) the account should be disabled (***disable_time_interval ERA***).

Note: At OSF DCE 1.1 level, the counter is only updated against invalid attempts at the master registry. Since the replicas do not communicate to the master, they cannot report invalid logins at a replica registry.

3.2.3.5 Password Management

The DCE registry maintains, in the standard policies, a password policy that allows you to set the following rules:

- Minimum password length
- Whether a password can be all spaces
- Whether a password can be all alphanumeric
- Password expiration date or lifetime

The first three items describe the password format. The standard password strength policies enforce these password rules, which might not be strong enough for many purposes. OSF DCE 1.1 includes the option to create a password management server that performs customized password checking (stronger rules) and generation. The password management server is invoked upon password creation or changes. The `pwd_strengthd` is an example provided to be used as a basis for a password management server that suits your needs.

A ***pwd_val_type ERA*** allows you to define whether an account password should be subject to extended password strength policies and whether it should be set by the user or be generated. If this ERA is zero or undefined for a principal, only the standard policies are applied. If extended password strength policies are set, then the ***pwd_mgmt_binding ERA*** defines a binding to a password management server. This server should run on the same machine as the security server.

When a password expires, the principal is locked out of DCE. You can allow a principal to log in with an expired password by setting the ***passwd_override ERA*** to 1:

```
dcecp> principal modify cell_admin -add {passwd_override 1}
```

With this setting, the regular `dce_login` (AIX) or `dcelogin` (OS/2 Beta) display an information message and accept the user without enforcing a change. On the other hand, if the DCE authentication method is used in AIX 4.1, this is the recommended setting, because the AIX login then requires the user to change the password. See also 3.8.1.1, "AIX and DCE Security Integration" on page 74 for more information.

3.2.3.6 Keys and Key Management

An important part of the authentication concepts of the DCE Security Service (or Kerberos) are keys and tickets. Keys are used to encrypt/decrypt messages. They are only known to a very limited number of principals. A ***principal's secret key*** is only known to the according principal and the Authentication Service. It is the principal's password encrypted by the Security Service runtime of the machine where the password is entered in clear text.

A ***conversation key*** is assigned by the Authentication Service (AS) and is *only* known by two principals engaged in a direct conversation. When server and client principal can understand each other using the conversation key, they can be sure of each other's identity because they trust that the AS which issued the key has properly identified each individual principal before.

A principal's secret key is primarily used to identify a principal to the AS. If the principal is a server, then its secret key is also used by the AS to encrypt a service ticket for access to this server. The client receives the ticket and passes it on to the server which can decrypt it. See also 3.2.3.7, "Tickets and Ticket Lifetimes" on page 59. After this initial sending of the ticket, only randomly generated conversation keys are used to avoid sending too many messages encrypted with the secret keys.

Users (interactive principals) log into DCE by entering their password. The login facility (3.2.3.1, "DCE Login Facility" on page 52) uses their password to generate their secret key. A non-interactive principal (such as a server) also needs a way to provide a password. The principal's key, generated from a password, is stored in a ***local keytab file*** on the machine where the principal resides. The password can be entered by the administrator or randomly generated. The randomly generated key is also sent to the registry.

The following commands add a randomly generated password to the bank_srv principal and forward the generated key to the registry:

```
# rgy_edit
rgy_edit> ktadd -p bank_srv -pw <bank_srv_password>
rgy_edit> ktadd -p bank_srv -a -r
```

This is the way to add a key in OSF DCE 1.0.x. In order to store a random password which is synchronized between the local keytab and the registry, a two-step procedure is necessary:

1. With the principal′s name and the plain password, as by specified by the administrator upon account creation, you need to create a first local keytab entry. The password is encrypted and stored in the keytab. A server principal is now ready to authenticate itself to the registry because the password is synchronized.
2. The ktadd procedure can now assume the principal′s identity, authenticate itself to the registry and change the password, on the principal′s behalf, to a random password. This random password is also stored in the local keytab file.

From now on, the server principal should regularly change its password to other randomly generated passwords to reduce the risk of being impersonated. The rgy_edit command is still available in OSF DCE 1.1, but it only lets you create a keytab entry on the local machine. The default keytab file is /krb5/v5srvtab on AIX and \OPT\DCELOCAL\krb5\v5srvtab on OS/2.

In OSF DCE 1.1, keytab management is performed via the DCE host daemon (dced), which allows management of a remote server host′s keytab. The dcecp keytab create command lets you create a ***dced keytab object*** in the dced namespace junction and associate it with a file. The following command creates a keytab object, /.:/hosts/ev1/config/keytab/bank_srv_tab, and associates it with the file /opt/dcelocal/keys/bank_srv_tab on host ev1:

```
# dcecp
dcecp> keytab create  /.:/hosts/ev1/config/keytab/bank_srv_tab -storage \
/opt/dcelocal/keys/bank_srv_tab -data {bank_srv plain 1 bank_srv_passwd}
```

A dced keytab object cannot be created without specifying a keytab entry. The command also fails if the /opt/dcelocal/keys directory does not exist or if the /opt/dcelocal/keys/bank_srv_tab object does already exist. The default keytab file, /krb5/v5srvtab, is associated with /.:/hosts/<hostname>/config/keytab/self. The following command can be used to list the contents of the newly created keytab from any host in the cell:

```
dcecp> keytab show /.:/hosts/ev1/config/keytab/bank_srv_tab
{uuid d88acdee-026c-11cf-b54a-10005a4f4629}
{annotation  {}}
{storage /opt/dcelocal/keys/bank_srv_tab}
{/.../cell1.itsc.austin.ibm.com/bank_srv des 1}
```

Once the dced keytab object with its associated disk file exists, more entries can be added. In order to create a random password for a newly created principal, you need to perform the same two-step procedure as with the rgy_edit command explained above. The following two commands perform this for the account atm_srv:

```
dcecp> keytab add /.:/hosts/ev1/config/keytab/bank_srv_tab -member atm_srv \
-key atm_srv_passwd -version 1
dcecp> keytab add /.:/hosts/ev1/config/keytab/bank_srv_tab -member atm_srv \
-random -registry
```

Long-running server applications usually spawn off a thread that checks the password expiration date, goes to sleep, wakes up shortly before the password expires, and updates the password with another randomly generated password.

The local Security Validation Service of the DCE daemon (sec_clientd or sclientd in DCE 1.0.x) automatically maintains the machine password in this way.

The Key Management API provides simple key management functions for non-interactive principals. See also 10.4.4, "Key Management and Secret Key Authentication" on page 193 for more information.

3.2.3.7 Tickets and Ticket Lifetimes

A ticket is a secret message issued by the Authentication Service (AS) to mediate a secure communication between a client and a server. The AS seals the ticket or parts of the ticket with the secret key of the server for which it is issued. The client receives the ticket and might be able to read some (non-encrypted) parts of it. However, if a part of ticket is not encrypted with the server′s key, there is an encrypted checksum which *seals* the ticket in such a way that the client cannot manipulate the ticket to its liking. The client then sends the ticket to the server, which can decrypt it. The ticket contains a conversation key which will be used to encrypted further messages between the two.

The ***Privilege-Ticket Granting Ticket (PTGT)*** is the client′s ticket to the Authentication Service (AS) which allows it to request ***service tickets*** to other DCE servers. The PTGT contains the EPAC of the client in readable form, but is sealed with a checksum. So, the client can read its own EPAC, but cannot manipulate it. Other tickets are usually completely encrypted.

Tickets have a lifetime (ten hours by default), after which they expire and are no longer useful. This value, defined as the ***default ticket lifetime*** property for the registry, can be changed. Principals can request tickets with a longer lifetime, which is gated by the lower value of either the time remaining on the principal′s TGT or the ***maximum certificate lifetime*** authentication policy of the registry or a specific account (default one day).

Tickets can be automatically renewed up to the time remaining on the principal′s TGT if the principal′s account is flagged to be ***able to get renewable certificates***.

A principal′s TGT lifetime can be requested via the login API when the principal establishes its own identity. For principals that log in with the interactive DCE login commands, the DCE Login Facility chooses to set the default ticket lifetime. This is why interactive user principals obtain a ten-hour TGT. To change this, you must change the *default ticket lifetime* property of the registry. The ***maximum renewable lifetime*** authentication policy defines the maximum TGT lifetime that can be requested.

When the TGT expires, a principal must reauthenticate. Users must log in again or run the `kinit` command. The `kinit` command allows the user to request a specific TGT lifetime. Long running servers that establish their own login context usually spawn off a thread that monitors the ticket expiration and reauthenticates in time. They do the same to monitor and refresh their keys (passwords) before they expire.

By increasing default and maximum ticket lifetimes, you could set up accounts so that tickets never expire. The shorter you make the lifetime, the greater the security of the system. On the other hand, extremely short ticket lifetimes cause frequent renewals and processing overhead.

Note: The *maximum renewable lifetime* and the *able to get renewable certificates* flag are not supported by the current OSF DCE 1.1 release. So, the maximum TGT is determined by the maximum certificate lifetime.

3.2.4 PAC and Extended PAC

When the user asks for a service, the privilege server extracts from the registry database all principal-related information and assembles a Privilege Attribute Certificate (PAC).

The OSF DCE 1.0 version of the PAC contains the following data:

- Authenticated — Indicates whether the PAC has been authenticated or not.
- Cell UUID — Identifies the principal′s cell.
- Principal UUID — Principal UUID and name associated with this PAC.
- Group UUID — Primary group UUID and name associated with this PAC.
- Number Local Groups — Number of the local groups in the pointer chain.
- Number Foreign Groups — Number of the foreign groups.
- Pointer Local Groups — Pointer to a chain of local groups.
- Pointer Foreign Groups — Pointer to a chain of foreign groups.

In OSF DCE 1.1, the PAC had to be extended to deliver the additional information that can be stored in the registry for each principal and to allow for delegation (see 3.2.5, "Delegation"). The Extended PAC (EPAC) contains the pre-DCE 1.1 PAC, the Extended Registry Attributes (ERAs) and information used to determine delegation initiator or targets.

The EPAC is sealed in the tickets that are used to establish client/server conversations and to check authorization of incoming requests.

An OSF DCE 1.1 server always needs an EPAC, whereas a pre-DCE 1.1 server can only deal with PACs. For a DCE 1.1 application server, the security runtime automatically converts a PAC delivered by a pre-DCE 1.1 client into an EPAC. For a pre-DCE 1.1 server, the security runtime automatically extracts the PAC data from the EPAC delivered by a DCE 1.1 client.

3.2.5 Delegation

Sometimes, as part of the execution of a remote request from a DCE client, a DCE application server needs to issue a request to another DCE server. The last server in the chain needs to authorize the request, but whose permission does the server check, the DCE client′s or the intermediary′s? In Release 1.0.x, the last server in a chain could not see the DCE client. Should the intermediary change its identity in order to act on behalf of the initiating client?

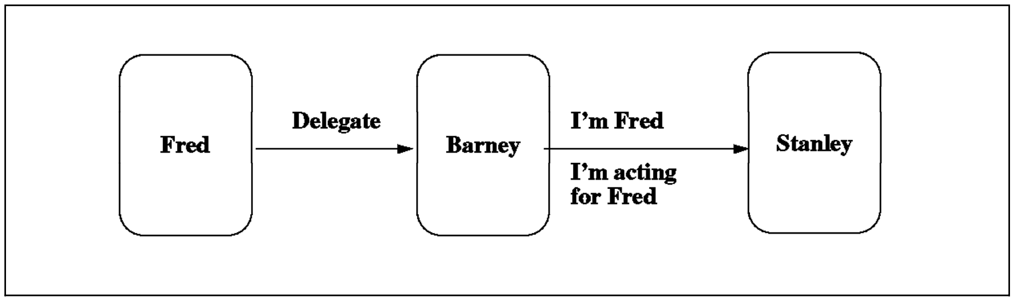


Figure 22. Delegation

As Figure 22 shows, the intermediary server can now pass on the client′s identity without having to change its own identity. It can either temporarily impersonate the initiator of the call (″I am Fred″) or pass on its own identity together with the initiator′s identity (″I am Barney and acting on behalf of Fred″). So there are two delegation types:

- ***Impersonation*** — The service ticket includes only the identity of the delegation chain initiator as well as target/delegate restrictions.
- ***Traced Delegation*** — The service ticket (also called the credentials) presented to the target server includes the identities of all members of the delegation chain. Each intermediary adds its own EPAC to the already existing credentials obtained from the predecessor′s ticket. This is the preferred type of delegation.

The DCE delegation model is based on the extension of two components which we will discuss:

1. Service tickets — Can contain multiple EPACs
2. ACLs — Entries for delegated access are added in the ACL model

The initiating client principal uses `sec_login_become_initiator()` to set the delegation type and to obtain a new login context enabled for delegation. The EPAC contains the type of delegation and target/delegate restrictions set by the initiating principal.

The client then starts authenticated RPC. Its PTGT is sent to the Authentication Service (AS), which prepares a service ticket to the first intermediary server. The client call is then routed to this server, which realizes that it needs to call another server to process the client′s call.

From the client′s service ticket, the first intermediary server learns the type of delegation and, depending on this type, calls `sec_login_become_delegate()` or `sec_login_become_impersonate()` to enable delegation on its part. The intermediary may specify additional target/delegate restrictions with one of these calls. Then it sends the ticket received from the client together with its own PTGT to the Security Service and requests a ticket for the next server.

The AS creates a new service ticket containing all the delegation information added up so far. The intermediary server presents this ticket to the next server, which may be the target server or another intermediary.

When a principal enables delegation or becomes an intermediary in a delegation chain, the principal may specify target and delegate restrictions. These are lists of principals which may become target servers in a delegation chain (***target restrictions***) and which may become intermediaries (***delegate restrictions***). If there is no list, all principals may become targets or delegates. If there is one or more lists set by the initiator and/or any intermediary, however, only the listed principals may become targets or delegates and only if they are listed in *every* forthcoming list.

When a target server is excluded from a list, the Security runtime replaces all principal identities which excluded this target server with the anonymous principal. Then the call is executed. However, it may fail to access an object because the anonymous principal might have insufficient permission.

When one or more intermediary server(s) are excluded from being delegate servers, their principal identities are replaced with the anonymous principal, and then the call is executed. So, the chain is not interrupted by target or delegate restrictions, but principal IDs may be replaced.

Part of the ticket that is used for contacting delegate servers is an encrypted ***delegation token*** which contains all the EPACs. In the case of delegation, the EPACs are also called an ***EPAC chain***. Once the request is received at the target server, this server evaluates the EPACs to check the permissions. If the application server uses an ACL Manager, it can leave the permission check up to the ACL Manager.

The ACLs in OSF DCE 1.1 contain ***new ACL Entry types***, such as user_obj_delegate, group_obj_delegate, user_delegate, and so on, to define the permissions of intermediary principals which act on behalf of the initiating principal. So, the object being protected by ACLs has to set up the according permissions if it wants to allow delegated access. See 3.2.6, "Access Control List Facility" for more information.

Pre-DCE 1.1 clients cannot request delegation and for pre-DCE 1.1 servers the EPACs in the chain are reduced to one PAC as described in 3.2.4, "PAC and Extended PAC" on page 60. For this case, the client has to pre-specify whether the initiator′s or the last intermediary′s EPACs are considered.

In order to put delegate authorization into objects defined in the Cell Directory Server, the directory on which we are making the change should have the CDS_DirectoryVersion attribute equal or greater than 4.0.

Note: Delegation is not just there in existing applications, it has to be built in by the application developers. The administrators then might be able to use an application-specific tool to enable delegation or to optionally define target/delegate restrictions and `acl_edit` to define permissions for the intermediaries, if necessary.

3.2.6 Access Control List Facility

All objects in the DCE namespace can have an associated ***access control list (ACL)*** that specifies which operations can be performed by which user. ACLs can be associated with files, directories, registry objects, or be implemented by arbitrary applications to control access to their internal objects.

Each ACL consists of multiple ***ACL entries*** that define who can access the object, what kind of access those principals or groups have to the object and what kind of access is allowed to unauthenticated users. The DCE ACL is a superset of POSIX ACL and conforms to the POSIX 1003.6 Draft 12. DCE ACLs implement a discretionary access control policy, where it is up to the user or the system administrator to set up the access policy for the objects.

This section gives a short explanation of ACLs and their management.

Note: The cell administrator or anyone who gains the cell administrator authority, legally or not, has access to any resource of the DCE cell.

3.2.6.1 ACL Managers and ACL Types

An ACL Manager implements ACLs for a specific set of objects that it controls. It defines the ***permission set*** and performs permission checks on behalf of application servers.

It is either the responsibility or the choice of an application developer to implement an ACL Manager to manage authorization. If they decide to do so, the ACL Manager has to support the calls defined in the RDACLIF API. This enables administration of the user-provided ACLs, through the ACL management tool `acl_edit` or the `dcecp acl` commands, which call this interface. In OSF DCE 1.1, the implementation of ACL Managers has been significantly simplified.

An application can implement different ACL Managers for different types of objects it manages. An example is the Security Service that currently implements several ACL Managers for its objects. See 3.2.1, "The Security Registry" on page 46 for a list of objects.

Applications that manage a hierarchical structure of objects, such as the DFS or CDS, may define different ACL types, such as:

- ***Object ACLs*** — This is the ACL of a specific object, which can be a container object (directory) or a leaf object (file).
- ***Initial Object Creation ACLs (IOC)*** — Define what ACLs a new leaf object in this container will inherit.
- ***Initial Container Creation ACLs (ICC)*** — Define what ACLs a new child-container will inherit. The ICC of the parent will determine the new container object ACL as well as the ICC of the child container. The child will inherit the IOC from its parent.

ACL types and ACL inheritance are illustrated in Figure 23 below.

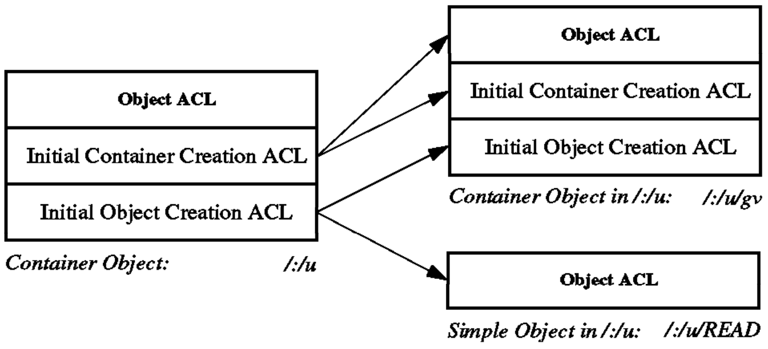

Object ACL
Initial Container Creation ACL
Initial Object Creation ACL
Container Object: /:/u
Object ACL
Initial Container Creation ACL
Initial Object Creation ACL
Container Object in /:/u: /:/u/gv
Object ACL
Simple Object in /:/u: /:/u/READ

Figure 23. ACL Inheritance

3.2.6.2 DCE ACL Entries

An ACL contains multiple ACL entries each of which defines the access rights of a specific principal (or group) to the object to which the ACL is associated. If the user does not match an ACL entry for a specific user, they might accrue the access rights granted to one or more groups, if they are a member of these groups.

An ACL entry consists of an entry type, an identifier dependent upon the entry type and a permission set:

- ***Entry type*** — The following types require no additional identifier because the grantee is either derived from the owner principal or is generally valid:

user_obj	The owner of the object.
group_obj	The primary group of the owner principal′s account.
other_obj	All principals in the default cell.
foreign_obj	All principals in a foreign cell.
any_obj	All local or foreign principals that do not match any other entry and all unauthenticated principals.
xxx*_delegate*	The ′xxx′ stands for all of the above entry types and is in effect when they are acting as delegates.

  The following types require an optional cell name and a group name or a user name:

user	A specific user of the default cell.
group	A specific group of the default cell.
foreign_user	A specific user of a foreign cell.
foreign_group	A specific user of a foreign cell.
xxx*_delegate*	The ′xxx′ stands for all of the above entry types and is in effect when they are acting as delegates.

  You can define two mask entries that define a maximum permission certain entry types can be granted.

mask_obj	The permission set of all entry types except for user_obj, user_obj_delegate, other_obj, and other_obj_delegate are ANDed with the permission set of this mask. This can be used to impose a temporary access restriction.
unauthenticated	All of the above entry types apply to authenticated users. If a user is unauthenticated, which means that their privilege attributes have not been certified by the Authentication Service, their permissions are masked (ANDed) with the *unauthenticated* mask. If there is no such mask, no permissions will be granted to unauthenticated users. Unauthenticated users that have no credentials at all are only checked against the any_other permissions which are then masked with the *unauthenticated* mask. Both the any_other entry and the *unauthenticated* mask, must be present for such users to get access; otherwise they will be denied any access.

- ***Identifier*** — Entry types that stand for specific users or groups require the name of the user or the group. ACL entry types for foreign users or groups require the cell name followed by the user/group name. For instance, the fully qualified name of user goofy in the cell xycorp.com would be *xycorp.com/goofy*.

- ***Permission set*** — Each ACL Manager can define a set of one-character permissions, where each character defines a certain access right. What access right a character stands for is ACL Manager-dependent. There are some conventions, though. When you access the ACL Manager of a specific object, you can display the defined permissions. For example, for the /.:/ev7_ch CDS object:

```
# acl_edit /.:/ev7_ch
sec_acl_edit> pe
Token   Description
r       Read entry attributes
w       Update entry attributes
d       Delete entry
t       Test attribute values
c       Change ACL
sec_acl_edit> exit
#
# dcecp
dcecp> acl pe /.:/ev7_ch
{r Read entry attributes}
{w Update entry attributes}
{d Delete entry}
{t Test attribute values}
{c Change ACL}
dcecp> exit
```

This is just a sample collection; there are many more permissions defined in the various ACL Managers.

3.2.6.3 Permission Evaluation

An ACL Manager searches the ACL entries in a predefined order, and if a principal matches a specific user_obj, user_obj_delegate, user, user_delegate, foreign_user, or foreign_user_delegate entry, it stops checking. Even if the principal were member of a group with more rights, it only gets what is in its user entry.

If there is no matching user entry, *all* group entries are searched, and all matching entries (groups of which the principal is a member) are ORed. If the requested right can be granted, checking stops.

If checking has not stopped yet, it checks whether the principal matches the other_obj and other_obj_delegate entries, then the foreign_other and foreign_other_delegate entries, and finally the any_other and any_other_delegate entries. Checking stops as soon as the principal requesting permission matches the criteria of one of these ACL types.

For initiator principals in ***delegation chains***, the xxx_delegate entries are not checked. These principals must match a standard ACL entry type. If the initiator principal is denied access, then checking stops. In traced delegation, all intermediaries are then checked according to the above evaluation procedure. All entries types are checked for them. They can match either a standard type or a xxx_delegate entry. The advantage of defining xxx_delegate entries for intermediaries over standard entries is that intermediaries are prevented from directly accessing objects (without being a delegate node). The final permission is the intersection of the permissions of the initiator principal and of each delegate principal. In other words, the initiator and each delegate must have the appropriate permission to be granted access.

If delegation or target restrictions have been applied (see 3.2.5, “Delegation” on page 60), then the anonymous principal might be present in the EPAC chain to be evaluated. The anonymous principal is represented by the UUID fad18d52-ac83-11cc-b72d-0800092784e9, not with a name. It qualifies for the other_obj, any_other, other_obj_deleg, and any_other_deleg entry types, if any of these are present.

The final permission is then masked with the mask_obj mask (except for user_obj, user_obj_delegate, other_obj or other_obj_delegate entries) and with the *unauthenticated* mask.

3.2.6.4 ACL Management Example

To define an ACL entry for an object, we can use the `acl_edit` command or the `dcecp acl` command. For example, to add read/write permissions to the principal nayeli for the /.:/APPS CDS directory and to deny any rights to principal gerardo, type the following:

```
dcecp> acl modify /.:/APPS -add  {user nayeli rw}
dcecp> acl modify /.:/APPS -add  {user gerardo}
dcecp> acl show /.:/APPS
{unauthenticated r--t---}
{user cell_admin rwdtcia}
{user hosts/ev4/cds-server rwdtcia}
{user nayeli rw-----}
{user gerardo -------}
{group subsys/dce/cds-admin rwdtcia}
{group subsys/dce/cds-server rwdtcia}
{any_other r--t---}
```

When principal gerardo requests access, the evaluation of his permission is stopped at his specific *user* entry. Because he has no permissions set, he is denied all access, even if he were a member of the subsys/dce/cds-admin group.

If a user without DCE credentials (one that did not perform a DCE login) comes in, they can acquire read and test permissions because it is specified in the any_other entry, and the *unauthenticated* mask allows this, too. If one of these entries were missing, they would not have any access at all.

3.2.7 Auditing

Originally, the three heads of Kerberos signified authentication, authorization and auditing. DCE′s security on releases 1.0.x does not provide auditing. Auditing is a new service provided by the OSF DCE release 1.1.

Once a principal has been authorized to perform an action, they might abuse their power for an unauthorized purpose. One common example is the Teller who, from every transaction performed on an account, sends the cents (or the fractions) to their account. Tellers are authorized to move money from one account to another; however, they are not usually authorized to take a percentage of each transaction for their own personal gain. Auditing can help to uncover such situations.

Auditing is necessary for detecting and recording critical events in distributed applications. The DCE Audit Service is a *non*-distributed service. Audit record logging is always done to file(s) on the machine where the audit API is used, that is on application servers or core servers. A log file is called ***audit trail file***. The audit daemon manages the *central audit trail file* of a machine. Audit clients can choose to use a *local audit trail file* which is directly accessed.

The DCE Audit Service has the following components:

- ***Audit daemon*** (`auditd`) — Performs the logging of audit events (audit records) into the central audit trail file and manages ***audit filters***. It must run on all server machines that use the machines′ central audit trail file and/or audit filters. The relevant audit information is provided by ***audit clients*** via an RPC interface.

- ***APIs*** — Allows audit clients to record audit events and the dcecp control program to control the audit daemon. These APIs can also be used to create tools that can analyze the audit records.
- ***Audit clients*** — Usually application servers and also DCE core services that are enabled for auditing. In OSF DCE 1.1, DTS and the Security Service are enabled as audit clients; so machines running one of these services should also run the audit daemon.

  Programmers of audit clients identify (RPC) functions that need to be audited, so called ***code points***. They assign an ***event number*** to each code point and use the DCE Audit API calls within each code point to record events. In other words, in each manager function of a DCE RPC server, the programmer uses `dce_aud_start()` to initialize an audit record, adds event information with the `dce_aud_put_ev_info()` calls and closes the record with the `dce_aud_commit()` call.

  Before adding records, an audit trail file has to be opened. This is achieved during server initialization through a `dce_aud_open()` call. When the DCE RPC server shuts down, it closes the audit trail file.
- ***Management functions*** — Administrators manage the operation of, and access control to, the audit daemon. They also manage the recorded events. The administrator obtains the event numbers from the developer and can define ***event classes*** by listing event numbers within event class files. Event classes group events with some commonality and facilitate management.

  Furthermore, the administrator can turn auditing on or off and define ***audit filters*** to limit such things as what event classes are recorded or which principals′ actions should or should not be monitored.
- ***Event Classes*** — Administrators can define event classes by logically grouping several events that have something in common. Event classes are defined in files in the /opt/dcelocal/etc/audit/ec directory in AIX and \OPT\DCELOCAL\etc\audit\ec directory in OS/2. The directory already contains some predefined classes for the DTS, Audit and Security Services.

  For class names, servers should use the format ***ec****_org_product_class* (vendors), such as ec_osf_dce_authentication, or the format ***dce****_server-name_class*, such as dce_sec_control. Since OS/2 is more limited on file names, the latter is called SECNTRL in OS/2.

  One event number can be used in several event classes. Classes can be used in filters to limit the recording of events.
- ***Filters*** — Filters can be used to limit the amount of records that are written to the audit trail. A filter consists of a ***filter subject identity*** and one or more ***filter guides***. The subject defines to whom the filter applies. For instance, it may apply to a specifically named principal, group, cell, or all (world). A guide defines a combination of an audit condition (event success, failure, access denied), an action (display and/or log to audit trail) and an event class(es) to which the filter will apply.
- ***ACL Manager*** — The permission to the audit daemon are set and modified with the ACL facility like any other object in the DCE namespace. To be able write audit records, an audit client principal must have *log* permission. To create filters, an administrator must have *write* permission. Using the dcecp

acl command on the /.:/hosts/*hostname*/audit-server object calls the audit daemon′s ACL Manager and lets you work with the daemon′s ACL.

Note: Auditing is not just there in existing applications, it has to be built in by the application developers. The administrators then might be able to use an application-specific tool to additionally configure auditing or to enable/disable it.

Note: Auditing is not supported in the current release of DCE for AIX Version 2.1. It will be enabled at a later date.

3.3 Intercell Authentication

The authentication mechanism is based on the exchange of secret messages between principals. We have explained that in 3.2.3.2, "Secret-Key Authentication Steps" on page 53. Principals trust each other in the same cell because the Authentication Services (AS) has authenticated both. This is true for all principals except the Authentication Service, which is the only principal in the cell which does not share its key. That is, the key of the AS is private, while the key of any other principal is secret (it′s known to two principals). The AS is trusted by all principals *a priori*.

When a principal in a cell wants to communicate with a principal in a foreign cell, it cannot obtain a ticket to the foreign principal from its local AS because the ticket to the foreign principal must be encrypted with the secret key of the foreign principal which is not known to the local AS. The secret key of the foreign principal is only known to the foreign principal itself and the foreign AS. Therefore, there must be a mechanism in place by which the two instances of the Authentication Service can securely convey information about their respective principals to one another, without having to share their private keys (sharing private keys would introduce an unacceptable security risk).

DCE solves this problem by extending the shared-secret key authentication model previously discussed. An administrator must create a principal representing the foreign cell in both Registries and they share the same key. The two registry database entries are known as ***mutual authentication surrogates***, and the two cells that maintain the mutual authentication surrogates are called ***trust peers***. It is through their surrogates that the two instances of the AS are enabled to convey information about their respective principals to one another, thus enabling a principal from one cell to acquire a ticket to a principal in another cell.

3.3.1 Intercell Authentication Steps

The intercell authentication process is similar to the one we discussed. When a client principal wants to communicate with a foreign server principal, the client′s security runtime recognizes by the name that the server is foreign and makes a request to the local AS for a TGT to the AS of the foreign cell. This request is called a Foreign TGT (FTGT) request or Cross-Cell TGT (XTGT).

The FTGT request proceeds like the request of any other TGT. The local AS constructs a ticket with the EPAC of the client and encrypts it using the secret key that the two authentication surrogates share. With this FTGT, the client requests a ticket to the foreign Privilege Service (PS) from the foreign AS. Because the FTGT is encrypted on the shared surrogate key, the foreign AS trusts the client and gives it a ticket to the foreign PS. The client then requests a Foreign PTGT (FPTGT) or Cross-Cell PTGT (XPTGT) from the foreign Privilege

Service. The FPTGT is simply the client′s original EPAC reencrypted with the key of the foreign PS. With the FPTGT, the client can request a ticket to any principal on the foreign cell.

If users have access to an account defined in the foreign registry, they can log in to that account by specifying the full principal name. For example, once the trust relationship is established between the two cells, a user in cell *itso7.austin.ibm.com*, with an account *gerardo* in the foreign cell *itso1.austin.ibm.com*, can log in as follows:

```
# dce_login /.../itso1.austin.ibm.com/gerardo
Enter Password:
#
```

On OS/2, the same cross-cell login would be:

```
[C\] dcelogin /.../itso1.austin.ibm.com/gerardo
Enter Password:
DCE LOGIN SUCCESSFUL
[C\]
```

3.3.2 Trust Relationship

The creation of mutual authentication surrogates is known as establishing a trust relationship between the two cells. This can be done using the `dcecp registry connect` command in OSF DCE 1.1 or with the `rgy_edit cell` command in OSF DCE 1.0.x. This command creates two accounts, one in your cell′s registry to represent the foreign cell and one in the foreign cell′s registry to represent your cell. Such an account is named *krbtgt/<cellname>*. Once the trust relationship is established, you can control foreign principals′ access to specific objects with ACL entries, just as you do for principals in the local cell.

There are two kinds of trust relationship that can be defined:

- ***Direct trust relationship*** — This is the trust relationship described above, where the two cell′s AS share a secret key. A direct trust relationship involves only two cells. In the context of hierarchical cells, we need to distinguish between two important variations of this relationship:
  - ***Direct trust relationship*** is between a parent and an immediate child cell of a cell hierarchy
  - ***Direct trust peer relationship*** is between any two cells that are not ancestors or descendents of each other
- ***Hierarchical transitive trust relationship*** — A transitive relationship involves three or more *hierarchical* cells. As shown in Figure 24 on page 70, the direct trust relationships between every parent-child pair build a ***trust path*** along the branches of the cell tree. All cells within a name hierarchy then trust each other when they can be reached following a trust path. This is called transitive trust.

  The trust path can go across *one* trust peer, but once it goes up and across, it cannot go up anymore. This means that a trust peer should be defined between the top-level cells of two cell hierarchies and not between any lower-level cells.

  Transitive trust drastically reduces the number of direct trust relationships that would have been necessary in OSF DCE 1.0.x to define any-to-any intercell communication with so many cells. On the other hand, obtaining an FPTGT going up and down a trust path involves dealing with many intermediary Authentication Services and introduces some processing overhead. After that, the foreign cell can be contacted directly to obtain service tickets as long as the FPTGT is valid.

A principal accessing a foreign cell using transitive trust needs not be authenticated by each cell traversed, only by the target cell.

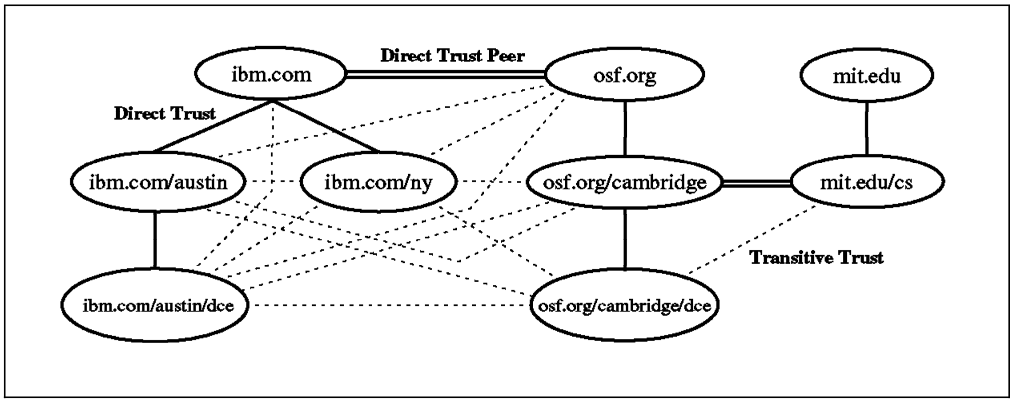


Figure 24. Direct and Transitive Trust Relationships

For example, Figure 24 shows different trust relationships. We have two direct *peer* trust relationships between ibm.com and osf.org and between mit.edu/cs and osf.org/cambridge. Within the cell hierarchies, we have a direct trust relationship between every parent and immediate child. This enables the transitive trust relationships shown with dotted lines.

See also 2.3.3, "Hierarchical Cells" on page 24 for more information about hierarchical cells.

Note: The transitive trust relationship is not supported on the OSF DCE Release 1.1 code; hence it is not supported in IBM DCE Version 2.1 for AIX and OS/2.

3.4 Security Administration Tools

The sec_admin command in AIX or the secadmin command in OS/2 are the security server administration tools used in OSF DCE 1.0.x. All the functionality of sec_admin, except for the monitor subcommand, is included in the dcecp command in OSF DCE 1.1. However, since dcecp is a complete and extendible shell script language (see Chapter 9, "DCE Control Program and Tcl" on page 147), the monitor function can easily be programmed.

The rgy_edit tool is used to manage the registry database. In OSF DCE 1.1, this command has been replaced by the dcecp command, but is still available for the time being. The only function not supported by the dcecp command is the maintenance of the local registry. For that, the rgy_edit command must still be used.

The ***local registry*** allows login from a machine if a network registry is not available. It contains account information and keys of principals that log into DCE on the local machine using the dce_login -c command. However, this command requires root user privileges in order to create a local registry entry. Otherwise, arbitrary users would be able to manipulate the local registry.

3.5 Security with RPC

Authentication, authorization and data protection are provided with the RPC runtime facility to enable applications to use the DCE Security Service for their RPC communication. Basically, RPC application servers define, during their initialization, what authentication, authorization and data protection levels they support. RPC clients may choose a security level they want to use. Of course, the level they choose must match a level supported by the server.

We describe authenticated RPC in the RPC chapter. Please see 10.4, “RPC and Security” on page 191.

3.6 Generic Security Service API (GSS-API)

The Generic Security Service (GSS) provides an alternative way of securing distributed applications that handle network communications by themselves. With the GSS-API, applications can establish secure connections and act like DCE RPC servers.

The GSS-API is a standard API for interfacing with security services, as defined by the IETF RFCs 1508 and 1509. It allows flexible use of the DCE security by programs, even if they don′t use DCE RPC to communicate (see Figure 25). Because the GSS-API is product neutral, you can define your security policy and implement it using the generic API. It will then, for instance, be portable between a NetSP and a DCE environment, which are (theoretically) pluggable and interchangeable underneath the API.

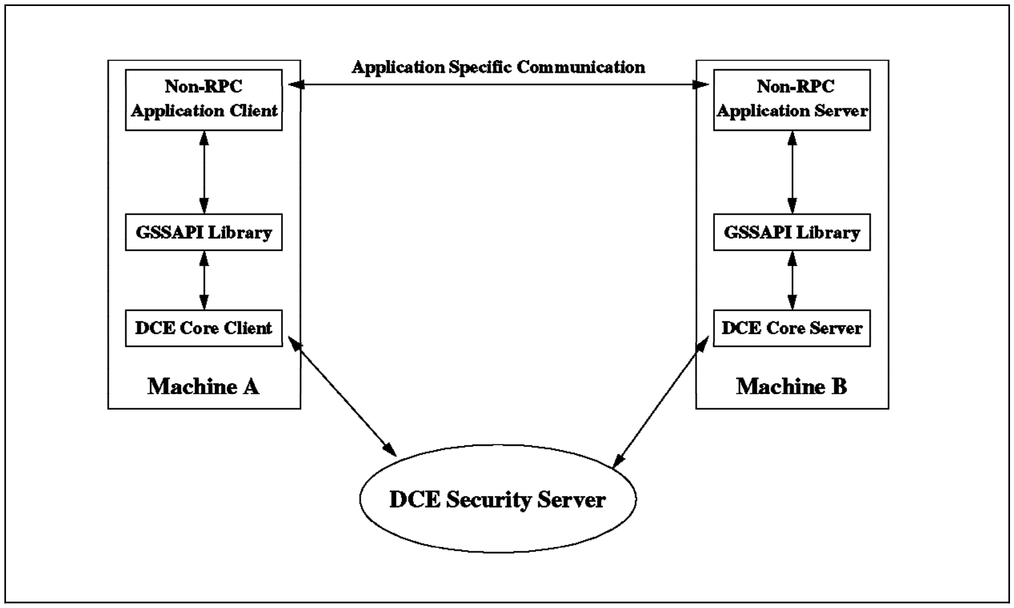


Figure 25. Generic Security Service API (GSS-API)

The GSS available with DCE includes the standard GSS-API routines (Internet RFC 1509) as well as OSF DCE extensions to the GSS-API routines. These extensions are routines that enable an application to use the DCE Security Service. However, if applications make use of the DCE extensions, they will not be portable to other security mechanisms because they will not understand, for instance, EPACs.

A GSS-API caller (usually the application client) accepts tokens provided to it by its local GSS-API implementation and transfers the tokens to a peer (usually the application server) on a remote system. That peer then passes the received tokens to its local GSS-API implementation for processing.

Clients as well as servers first have to authenticate themselves with the security server (network login). Clients then use the `gss_init_sec_context()` call to create the token needed to access a server. In the DCE environment, this call will result in a request to the privilege service for a service ticket to the specified server by presenting the client TGT (Ticket Granting Ticket) obtained from the login. This ticket contains the DCE Extended Privilege Attribute Certificate (EPAC) and is returned to the client within the GSS-API security context token. The client sends the token to the server, which hands it over to the GSS-API. The server-side GSS-API verifies the client security context and keeps it for later references until it is destroyed.

In addition to exchanging tokens for initial authentication, the application (client or server) can then create and validate tokens, so called signature tokens, to protect messages. The sender of a message requests a signature token from the GSS-API and sends the token together with the message. The receiver calls the GSS-API to verify the signature token along with the message.

A typical GSS-API caller is itself a communications protocol, calling on GSS-API in order to protect its communications with the underlying security mechanisms.

3.7 DCE Security and Other Core Components

In the following two sections, we briefly mention the effects of the Security Service on two other DCE core components: the DTS and the CDS.

3.7.1 DCE Security and DTS

Time is defined with tickets, which have a lifetime with a starting and an ending time. If the network time skew factor becomes too big, it might happen that unexpired tickets may be regarded as invalid, and expired tickets may be considered valid. Time in DCE is very important. This is because from the security viewpoint, time can open some security holes in DCE. Time should always be under the control of the cell administrator.

If you attempt to run DCELOGIN from DOS Windows or OS/2, you might get an error message such as:

```
Password validation failure, - Clock skew too great (dce/krb)
```

If the DCE security server and the DOS Windows client are not synchronized, the skew should be less than five minutes.

3.7.2 DCE Security and Naming

The security namespace is rooted in the CDS namespace; therefore security names have an equivalent CDS name. However, the Security Service refers to its objects in a different manner than commands, which are able to operate on objects of different name spaces, such as the `acl_edit` or `dcecp acl` commands.

All security commands, such as `dce_login` and `rgy_edit`, use and (only) understand names of the security namespace, such as the following principal name:

```
/.../itso7.austin.ibm.com/nayeli
```

What might puzzle you is that this name looks like an object in the CDS namespace. However, for commands that work on objects of the global namespace, such as `acl_edit`, the above name is undefined:

```
# acl_edit /.../itso7.austin.ibm.com/nayeli
ERROR: acl object not found (dce / sec)
Unable to bind to object /.../itso7.austin.ibm.com/nayeli
```

The `acl_edit` expects the principal's fully qualified object name in the CDS namespace:

```
# acl_edit /.../itso7.austin.ibm.com/sec/principal/nayeli
sec_acl_edit>
```

This command allows you to manage permissions on the registry object, such as who is allowed to delete this principal.

The naming scheme becomes even more confusing when the principal provides an RPC server with its own ACL Manager, such as the DCE daemon with the principal name *hosts/ev7/self*. The global name /.../itso7.austin.ibm.com/hosts/ev7/self stands for the principal name as used by the security commands and for a CDS object as used by RPC to advertise binding information. So, the name is context sensitive and contains three different sets of ACLs:

1. `acl_edit /.../itso7.austin.ibm.com/sec/principal/hosts/ev7/self`

   This ACL controls who may access and manipulate the principal definition in the registry. The command accesses the registry object.

2. `acl_edit /.../itso7.austin.ibm.com/hosts/ev7/self`

   This command uses the binding information stored in the CDS object to contact the DCE daemon and its ACL Manager. It allows you to define the permissions to manage the DCE daemon.

3. `acl_edit -e /.../itso7.austin.ibm.com/hosts/ev7/self`

   This command accesses the CDS object entry and defines who may manipulate it; for example, who may add a new binding handle to it or delete it from the namespace.

3.8 Platform-Specific Implementation

This section describes platform-specific implementation differences of the Security Service and features that are not available on all platforms.

3.8.1 Security Service on AIX Version 4

IBM DCE for AIX Version 2.1 is the implementation of OSF DCE 1.1. However, for several reasons, some of the Security Service features of OSF DCE 1.1 described in this document did not make it into the current release, such as:

- Transitive Trust (not in the OSF release)
- Renewable Tickets (not in the OSF release)
- Auditing (not in DCE for AIX Version 2.1)
- Hierarchical Cells (CDS; not in DCE for AIX Version 2.1)

In the rest of this section, we will discuss some issues present in all UNIX operating systems. Special considerations are necessary because UNIX is a multi-user operating system with an already implemented security system that must be integrated, or at least coexist, with the DCE security.

3.8.1.1 AIX and DCE Security Integration

In the DCE for AIX Version 2.1, the AIX base operation security services have been integrated with the DCE Security Services. In previous versions of DCE, the user had to log into AIX first and then to DCE. This required the maintenance of two user IDs and passwords. Furthermore, the DCE login credentials (represented by the environment variable KRB5CCNAME) could only be passed to child windows, but not to parent windows or other descendents of the parent window. Depending on your environment, you might have had to log into DCE several times.

This release of DCE permits the user to see a single-system image rather than separate images of AIX and DCE. Most users will be able to acquire DCE credentials through AIX commands, such as `login` and `su`, to change their password through the AIX `passwd` command and to get information from the DCE registry through the standard C library (libc.a) routines, such as `getpw*()` and `getgr*()`. Remote telnet or ftp users will also authenticate with the DCE registry when they access an account set up for DCE authentication.

To support DCE integration with the AIX login, two new user attributes (*SYSTEM* and *registry*) have been defined in the /etc/security/user file. This file defines such things as authentication methods, password policies or the umask for the user accounts of the local system. It defines default values and allows you to create a stanza for each individual user that may override some or all of the default values. The file also contains lots of explanations.

The ***SYSTEM attribute*** is used to select one or more of the system-provided methods which authenticate a user to the local machine. The valid values are boolean expression strings that may contain the boolean operators *AND* and *OR*. The valid values are:

`files`	Use local authentication with the /etc/security/passwd file
`compat`	Use local files and/or NIS
`DCE`	Authenticate through DCE

For example, to use DCE Security Service for authentication when a user logs in or, if DCE is not available, local files or NIS, we can define the following entry in the /etc/security/user:

```
auth1 = SYSTEM
SYSTEM = ″DCE OR (DCE[UNAVAIL] AND compat)″
```

The SYSTEM attribute is used in the *auth1 or auth2* attributes to specify that a system-provided authentication method is to be used. If this is the default definition, users with no local definitions can log in. Such users are called *wanderers*. However, if the network is down, wanderers will not be able to log in.

The ***registry attribute*** is used to force password operations to be performed either locally or to the DCE Security Service. It defines the database in which a user′s password is administered or where password queries and changes take place. The valid values are:

`files`	Use the local /etc/security/passwd file
`NIS`	Use NIS
`DCE`	Use the DCE registry

It is strongly recommended to maintain the DCE registry and local files synchronized as closely as possible. This means that for a particular user or group, the user ID and group ID should be the same on all systems of the cell and in the registry. If this is not the case, a user authenticating with DCE and assuming the user ID stored in the registry might appear as another user on the local machine.

For users that have a different user ID in the DCE registry, you should set the *SYSTEM* and *registry* attributes in the /etc/security/user file to *local* (or *compat/NIS*). This protects local resources from unauthorized access. You can also use the password_override file to map incoming DCE principals to other local account attributes or deny them access at all. See 3.8.1.2, “The passwd_override File” below.

To activate the security integration, you must do the following:

1. Ensure that the ***/usr/lib/security/DCE*** module is installed on the system.
2. Edit the ***/etc/security/login.cfg*** file, and add the following lines:

   ```
   DCE:
      program = /usr/lib/security/DCE
   ```

3. Ensure that the ***dceunixd*** daemon is running. You can add this daemon to the /etc/inittab file. This daemon communicates to the DCE servers secd and dced on behalf of the AIX commands. Be sure to start DCE services before dceunixd.
4. Edit the ***/etc/security/user*** stanza file to define the authentication method(s) for the users. This means setting the ***auth1*** and ***SYSTEM*** attributes as explained above.
5. To explicitly prevent certain DCE users from any access to the local system, you should modify the /opt/dcelocal/etc/passwd_override and /opt/dcelocal/etc/group_override files.

For users that authenticate via DCE, you should set the passwd_override ERA to 1, so that DCE allows them to log in with an expired password. The AIX login process then requires the user to change the password. Otherwise the user would be locked out and the cell administrator would have to set a new password. See also 3.2.3.5, “Password Management” on page 56.

3.8.1.2 The passwd_override File

The local administrator can deny DCE authenticated access to the local machine for a specific principal by putting the appropiate entry in the /opt/dcelocal/etc/passwd_override or /opt/dcelocal/etc/group_override file. For example, to deny local system access to DCE user joe, we can add the following line to the passwd_overrride file:

```
joe:OMIT:::::
```

The fields in the passwd_overrride file correspond to the fields you find in an /etc/passwd file. The override file can also be used to map account attributes defined in the DCE registry to different local attributes when that user logs into the local system with their DCE identity. So, this file can be used to locally adjust unsynchronized user IDs.

3.8.1.3 The passwd_import and passwd_export Tools

The passwd_import command is a mechanism for creating registry database entries from local password and group file entries. If there are duplicate entries, passwd_import follows your directions on how to handle them.

The /opt/dcelocal/bin/passwd_export command creates local password and group files from registry data. Use passwd_export to keep these local files consistent with the registry database. When passwd_export runs, it makes backup copies of the current password and group files, if they exist. The files are named passwd.bak and group.bak, respectively. The passwd_export is commonly run through an entry in root′s crontab file.

3.8.1.4 AIX and DCE ACL Interoperability

The editor acl_edit is used for updating DCE ACL. The editor works via RPC; so the administrator can be local or remote. The acl_edit must not be confused with the acledit command, which is the ACL editor for AIX ACL. The two ACLs are not compatible.

3.8.1.5 ACL Awareness

It is important to understand that all UNIX commands, such as tar, cpio or dd, do not preserve DFS ACLs. So, it is a good rule to have some sort of migration tool available that helps you backup and restore ACLs on the objects if you need to use AIX commands and want to preserve the ACLs. To store the ACL of an object into a file and reinstate them from a file, use the following commands:

```
# acl_edit <object> -l  >  <object>.acl
# acl_edit <object> -f <object>.acl
#
```

The DFS backup and restore commands preserve ACLs for DFS, but they cannot be used when you want to move files from one cell to another.

The AIX chmod, chown, chgrp commands also affect the DFS ACLs. They will recalculate file ownership and permissions for the user_obj, group_obj and other_obj.

3.8.1.6 DCE Security Integrity

It would be good system administration practice to add to the DCE security daemon and command to the local AIX Trusted Computing Base (TCB) and to the */etc/security/sysck.cfg* database. This will allow the system administrator to monitor the integrity of the DCE security sensitive files and commands and verify that they have not been compromised. There is an installation option for the TCB when installing AIX 4.1. TCB cannot be added after AIX 4.1 is installed.

First of all, it is important to add all the DCE security commands to the local AIX TCB with the chtcb command as follows:

```
# chtcb on /usr/lpp/dce/bin/*
# chtcb query /usr/lpp/dce/bin/dce_login
/usr/lpp/dce/bin/dce_login is in the TCB
```

After having done that for each command in the /usr/lpp/dce/bin directory, add an entry to the /etc/security/sysck.cfg command as follows:

```
/usr/lpp/dce/bin/kinit:
        class = inventory,apply
        owner = bin
        group = bin
        mode = TCB,555
        symlinks = /usr/bin/kinit
        type = FILE
        checksum = "14226   210"
        size = 214231
```

Remember, the checksum for the file is calculated with the sum command as follows:

```
# sum -r kinit
14226   210 kinit
#
```

You must also add the */krb5* and */var/dce/security* directory and files to the TCB and the *sysck.cfg* file. Remember that the registry database, /var/dce/security/rgy_data/rgy, and all the files under /var/dce/security/rgy_data have both checksum and size VOLATILE.

Refer to the tcbck command to add entries in the sysck file and to verify the integrity of the DCE security sensitive files.

3.8.2 Security Service on AIX Version 3.2

IBM DCE for AIX Version 1.3 runs on AIX Version 3.2 and provides a full implementation of OSF DCE 1.0.3. The following sections, that were described above for AIX Version 4, are also valid for AIX Version 3.2:

- 3.8.1.2, "The passwd_override File" on page 75
- 3.8.1.3, "The passwd_import and passwd_export Tools" on page 76
- 3.8.1.4, "AIX and DCE ACL Interoperability" on page 76
- 3.8.1.5, "ACL Awareness" on page 76
- 3.8.1.6, "DCE Security Integrity" on page 76

The security integration as described in 3.8.1.1, "AIX and DCE Security Integration" on page 74 is not available in AIX 3.2.5. If the user wants to get their DCE network credentials, they have to log in as a normal AIX user and then use the dce_login command, providing their DCE user name and DCE password. This will launch a new shell with the KRB5CCNAME environment variable set to the path name of the credential file for that user.

The user can then use line-oriented commands like klist, kinit, and kdestroy, respectively, to see or renew their credentials or to destroy them.

It is possible to have better integration of DCE authentication in AIX with a product named Single Login/6000. It allows users to log in with their DCE name and password and to automatically acquire their network credentials. The user does not have to be defined on any local machine; their local definitions are automatically generated and maintained. The CDS is used to keep track of where the user is currently logged in. It contains a password management function, can invalidate a user account if too many login attempts fail and can control the number of times a user can concurrently log in. It also implements a department concept. Departmental users can log into any machine in their department; global users can log into any machine in the cell.

For more information (more details, latest versions, ordering, prices), contact the following address:

```
IBM Deutschland Entwicklung GmbH
Customized Banking Technology
att. Hans-Juergen Dittgen
     Schoenaicher Str. 220
     71032 Boeblingen
         Germany
     Tel.  xx49-7031-16-4867
     Fax   xx49-7031-16-4572
     IBMMAIL: DEIBMQVR at IBMMAIL
     Vnet: DITTGEN at BOEVM4
```

This product is also discussed in more detail in the redbook *Using and Administering AIX DCE 1.3*.

If you are working for IBM, Single Login/6000 is available on the AIXTOOLS disk. To obtain a copy, type the following command from the VM command line:

```
TOOLS SENDTO USDIST MKTTOOLS AIXTOOLS GET SI_LOGIN PACKAGE
```

3.8.3 Security Service on OS/2 Warp

The current version of DCE for OS/2 is at OSF level 1.0.2. IBM DCE for OS/2 Warp Version 2.1 is available in Beta and is an implementation of OSF DCE 1.1. The Beta version currently contains the same restrictions as listed in 3.8.1, "Security Service on AIX Version 4" on page 73.

Security in OS/2 is identical to AIX DCE Security in terms of security components, APIs and security administration tools. The rgy_edit, acl_edit security administration tools are available on this operating system. The sec_admin command is called secadmin, and dce_login is called dcelogin in OS/2 DCE Version 2.1. All the key management commands, kinit, klist and kdestroy, are available. On OS/2, security can be configured either as a client or as a server.

The dcelogin command has been enhanced to support a systemwide login. The systemwide login (option *-s*) allows a principal to log into a given session and to have the resulting DCE credentials be associated with all DCE programs subsequently started in that session and all other sessions. Also, DCE 2.1 includes a new command, dcelgoff, to be used in conjunction with the systemwide login. It prevents DCE programs that are subsequently started from inheriting credentials established by a previous systemwide login.

3.8.4 DOS Windows DCE Security

The current DCE for DOS Windows is at OSF DCE level 1.0.2. Security can be configured only as a client. The administration tools, rgy_edit and acl_edit, are available. The key management kinit and kdestroy and the dce_login command are available to participate in a DCE cell. All these commands are available through the Windows presentation manager at this time. Also, the implementation of OSF DCE on DOS Windows provides the DCE security API.

Chapter 4. Distributed Time Service

Many applications use timestamps to coordinate independent events. Although not mandatory, the Distributed Time Service (DTS) should run on every host in the Distributed Computing Environment, keeping host clocks closely synchronized. It is also possible to synchronize the distributed environment with others by connecting to an external time signal, such as a *Universal Time Coordinated (UTC)* provider.

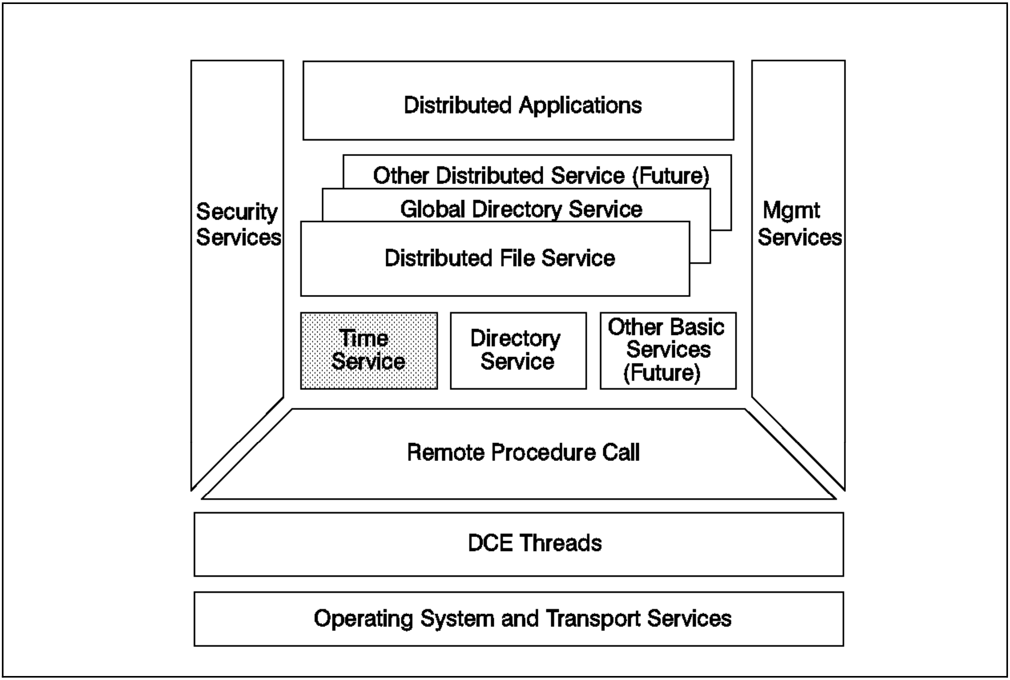


Figure 26. Distributed Time Services as a DCE Component

The components of DTS are:

- Time clerk
- Time servers
  - Local time server
  - Global time server
  - Courier time server
  - Backup courier time server
- DTS application programming interface (DTS API)
- Time provider interface (TPI)
- Time format, which includes an inaccuracy value

The active components are the time clerk and the time servers. The two interfaces, DTS API and TPI, are the developing interfaces for the DTS.

4.1 Why a Time Service?

Problems can occur when clocks are not synchronized. On networks composed of multiple hosts, each host has its own clock and its own time reference.

A good example of troubles due to unsynchronized clocks is the make command. The make command selects files for compilation based on their creation time.

Modified source files on a tardy host can appear older than corresponding object files on the host with the faster clock. In this case, make does not recompile files that should be recompiled.

DTS helps to avoid such situations by synchronizing host clocks in LANs and WANs with the following:

- DTS provides a way to periodically synchronize the clocks on the different hosts in a distributed environment.
- DTS also provides a way of keeping that synchronized notion of time close to the *correct* time. In DTS, a *correct* time is considered to be *Universal Time Coordinated (UTC)*, an international standard.

4.2 DTS Daemon

The DTS clerk and DTS server are both implemented as a DTS daemon (dtsd) process. They are therefore sometimes denoted with the neutral term DTS entity. A dtsd process becomes a server in response to a management command. Each of these processes communicate with each other via authenticated RPC.

An administrator may issue commands to a dtsd via the dtscp command (DCE 1.0.x), which is still available, or with the dcecp command (DCE 1.1). A dtsd makes RPC calls to other DTS daemons configured as servers.

The dtsd command restarts the DTS daemon. With the option dtsd -c, the DTS daemon is started as a time clerk and with the option dtsd -s as time server. When the system is booting or restarting DCE, this command is automatically executed as part of the overall DCE configuration procedure.

We might use dtsd interactively only when troubleshooting.

4.2.1 Configuration Parameters for DTS Daemons

The DTS entities do not have a configuration file or database. However, they have many configuration parameters that influence their behavior. These values have to be provided to the running dtsd process by means of the dtscp or the dcecp command. To list all configuration values, enter:

```
#dtscp show all
```

or

```
#dcecp
dcecp> dts show
dcecp> dts show /.:/hosts/ev1/dts-entity
```

Note: The dcecp allows dealing with remote DTS entities, whereas dtscp only affects the local system.

These attributes define the *type* of the DTS entity (server or clerk) and, for servers, their role. By changing these values, for instance, a clerk can be temporarily converted into a server.

The settings of these values also affect the accuracy of the system time and the network load. The skew factor, the difference between any two system clocks, can be made smaller through frequent synchronizations with a high number of servers. However, this obviously generates more traffic on the network. A tradeoff is necessary.

Note: The configuration parameters are lost when the DTS entity is stopped. They have to be set manually over and over again, or the appropriate commands have to be included into the startup procedure to effect a permanent change. Also, check the default values indicated in this publication with the `dcecp dts show` command. They might have changed in the final product.

4.2.2 The Required Number of Servers

The *Servers Required* (DCE 1.0.x) or *minservers* (DCE 1.1) attribute specifies how many DTS servers must supply time values before the local clock can be synchronized.

You can change this attribute for each running DTS daemon with the following command. For example, to change the number to four, enter:

```
dcecp> dts modify -minservers 4
```

Since configuration parameters are volatile and on a per-system basis, this attribute has to be set on each system every time the DTS entity is restarted.

Note: If the DTS entity does not get a sufficient number of time values (default is 1 for clerks and 3 for servers), it never does adjust its clock. So, special care has to be taken that either the required number of servers can be reached from every system, or the attribute has to be adjusted to match the cell configuration. This is particularly important for servers. If servers do not synchronize, their clock might drift away from the other systems, and even worse, clients might pick up that faulty time when their *minservers* attribute is 1.

4.2.3 DTS Clerk

The time clerk is the client side of the DTS. It keeps the machine′s local time synchronized by asking as many time servers as is specified in its *minservers* attribute. The default value is 1. In this case, the clerk randomly picks one server and adjusts its clock to that value, no matter how different this time value might be from its own system time.

The clerk synchronizes every 10 minutes, if necessary. Its *syncinterval* (DCE 1.1) defaults to 10 minutes. In DCE 1.0.x, this parameter was called *Synchronization Hold Down*.

It is recommended to increase the *minservers* value to three if there are at least three servers in the cell to make sure a faulty server does not mess up clerk-only systems. Should network performance become a problem, it is better to increase synchronization intervals than to rely on just one DTS server.

See 4.3, "How Does Clock Synchronization Work?" on page 84 for more details on how clocks are synchronized.

4.2.4 DTS Servers

A time server is a node that is designated to answer queries about time. The number of time servers in a DCE cell is configurable; at least three per LAN is a typical number. Time clerks query these time servers for the time, and the time servers query each other, computing the new system time and adjusting their own clocks as appropriate.

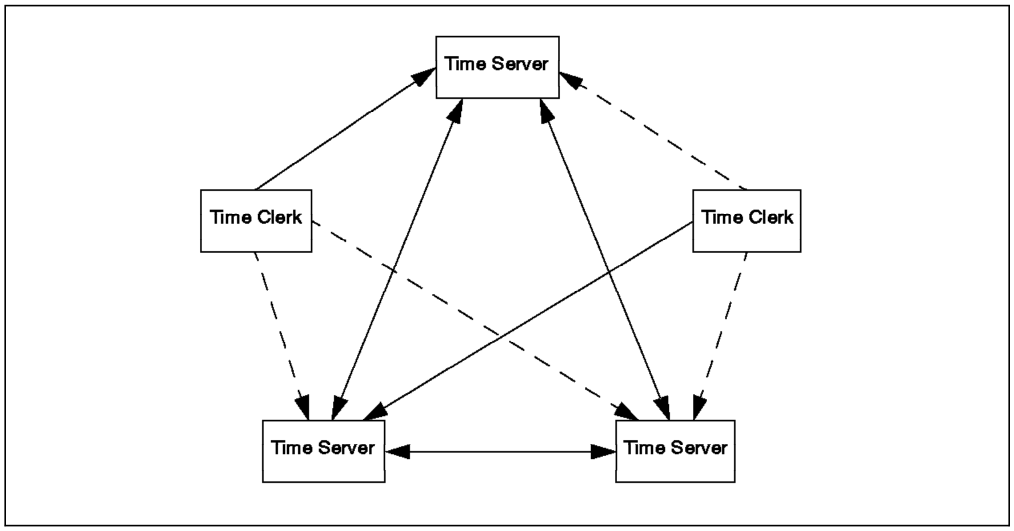


Figure 27. Time Servers and Clerks Requesting Time Values. In the default configuration, the three servers query each other, and each client randomly queries one server at each synchronization.

If an external time provider is present (see 4.5, "External Time Providers" on page 91) on a DTS server, this server only synchronizes with the time provider. If not, a server synchronizes with other servers.

The `minservers` attribute value defaults to 3 for servers. The server′s own time value is one of them. If there are not two other time servers available, they don′t synchronize, and their clocks might drift apart from each other. The synchronization interval is two minutes by default, and you can modify this default. To show the attributes, use the following command:

```
dcecp> dts show
 ...
{syncinterval +0-00:02:00.000I-----}
 ...
```

If you want do change this value, for example, to five minutes, use the following command:

```
dcecp> dts modify -syncinterval +0-00:05:00.000
```

Note: This change modifies the current `dtsd`. After a reboot, the DTS process will be reset to the default value.

When a DTS server is configured, its name is entered into the `/.:/lan-profile` of its own LAN. By synchronizing among each other, DTS servers maintain an accurate time in an entire cell with minimal skew factors between every system. To make synchronization work across LAN boundaries, DTS defines DTS server roles which are not mutually exclusive: local-only, global and three courier types.

4.2.4.1 Local Time Server

A collection of computers that are close in terms of communication delay (a LAN) needs a set of *local time servers* to keep the time. These servers periodically synchronize their clocks with one another.

The local time servers are in the same LAN and cannot be reached or used from the outside. When the required number of servers is not on the LAN, each DTS entity must contact as many global time servers as is necessary to get the required number of values. To find a global time server, the dtsd performs a lockup in the CDS.

4.2.4.2 Global Time Server

The *global time servers* are time servers that advertise themselves in the /.:/cell-profile in CDS. This is what makes the difference to local time servers and is why they are globally accessible.

A global time server acts as a local server for DTS servers and clients in the same LAN. Global servers are used by DTS couriers from other LANs within the cell. This feature helps keep multiple LANs within a cell synchronized while minimizing network traffic across expensive or slow WAN links.

If clerks and local time servers cannot get the required number of servers in their own LAN, they must contact global time servers, if there are any.

Note: A DTS server can either be configured as local or global. However, global servers are also local servers in their own LAN.

4.2.4.3 Courier Time Server

Each local and global time server has a courier role assigned. There are three roles:

- Courier
- Backup (courier)
- Noncourier

A *courier time server* must synchronize with one global time server outside the courier's LAN even if it had enough local time servers to query. It thus imports a LAN-external time by synchronizing with an outside time server.

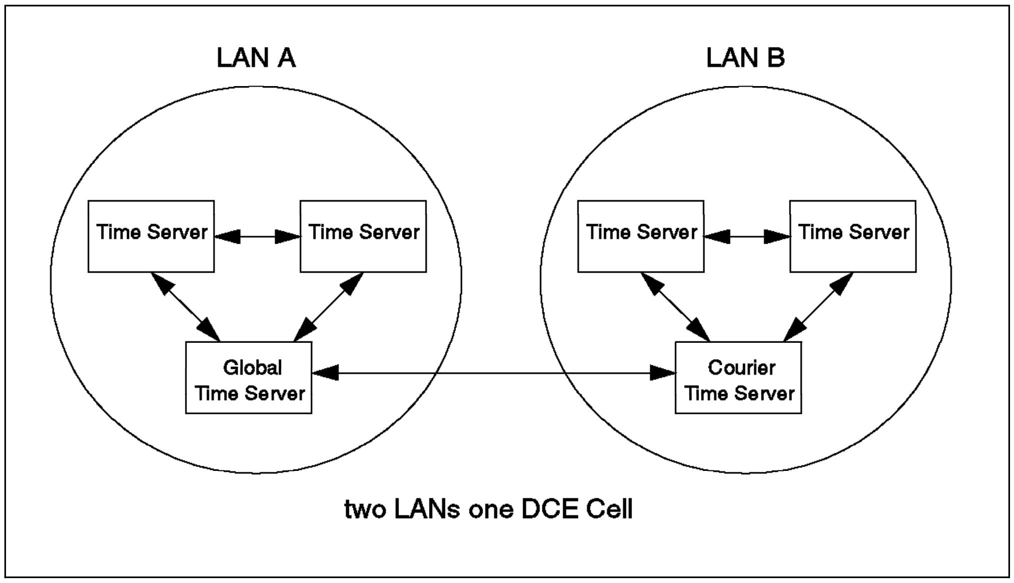


Figure 28. Local, Courier and Global Time Servers

In Figure 28, the courier time server in LAN B must query the global time server on LAN A. In this configuration, the LAN B synchronizes its time to the time in LAN A. If you want a mutual synchronization between both LANs, you need a courier and a global server in both LANs.

Noncouriers are released from the obligation to ever contact a global server as long as they can reach enough local time servers. This is the default.

Backup couriers have to be ready to stand in for a courier if the courier becomes unavailable. How do they know when they are on? Every DTS server maintains an in-memory list of local DTS servers with their attributes. So, they know who the courier is and whether it is available. If the courier becomes unavailable, the backup couriers start a negotiation process to elect a new courier.

4.3 How Does Clock Synchronization Work?

The DTS entity (daemon) running on every system is responsible for obtaining the *correct* time from a number of time servers and for adjusting its own clock. This process is basically the same for servers and clerks, except for the fact that a server may have an external time provider and that a server also uses its own time for the calculation of the correct time. This section explains how this is done.

4.3.1 Time Intervals, Inaccuracy, Synchronization Triggering

DTS uses an inaccuracy value, or tolerance, to determine the relative precision of time values that it obtains from system clocks and external time providers. The time value obtained from a DTS entity is a time interval that contains the correct time, rather than a point on a continuum.

The measurement of the inaccuracy value takes into account cumulative clock error, communications delays and processing delays. System clocks continuously *drift* away from the correct time in the order of a few seconds per day. DTS uses manufacturers′ specifications to calculate the amount of time the clock may have drifted since DTS previously read the clock. This amount is added to the previously existing inaccuracy.

So, the inaccuracy of the local system clock, the time interval, is constantly increasing. If the inaccuracy value reaches a certain threshold, the *maxinaccuracy* value, a synchronization is triggered. The default for this value is 0.10 seconds (100 milliseconds). However, if the amount of time passed since the last synchronization is less than the *syncinterval* (DCE 1.1), synchronization is suppressed until the synchronization interval is used up. In DCE 1.0.x this parameter was called the *Synchronization Hold Down*, which more accurately describes its purpose: It prevents too frequent synchronizations.

A successful synchronization results in a more accurate time value with a smaller *skew* to the other systems in the cell and a smaller inaccuracy interval.

4.3.2 Getting the Correct Time

Each DTS daemon uses the LAN profile to find DTS servers in the same LAN. The `/.:/lan-profile` is the default LAN profile used by DTS. This profile contains entries for the local DTS server set, meaning all servers on the LAN. Global servers are listed in the `/.:/cell-profile`. The DTS daemon maintains an in-memory list of all local and global DTS servers with some of their attributes. So, it knows, for example, their epoch, their courier role and whether they are faulty.

This in-memory list about DTS servers is updated at every synchronization by querying CDS for time servers and their bindings, which, by the way, may create considerable network traffic and CDS accesses. In this way, new servers are detected on time for every synchronization. The list is purged every two hours, which gives faulty or bad-epoch servers a chance for rehabilitation.

The DTS daemons randomize the internal list of servers to choose the right number of servers that are not marked faulty. If the DTS daemon is itself a server, it only needs *minservers-1* servers and only chooses servers with the same epoch number. Each time the synchronization is done, it will randomize the list of servers. This is an attempt to ensure that all servers in the cell will be used to perform the synchronization, unless they are marked faulty.

Each server replies with what it estimates to be the correct time and also with a bound on the accuracy of this time estimate. Thus, the DTS daemon receives a set of intervals with which it has to determine a current time interval that is tighter than any intervals obtained. See Figure 29.

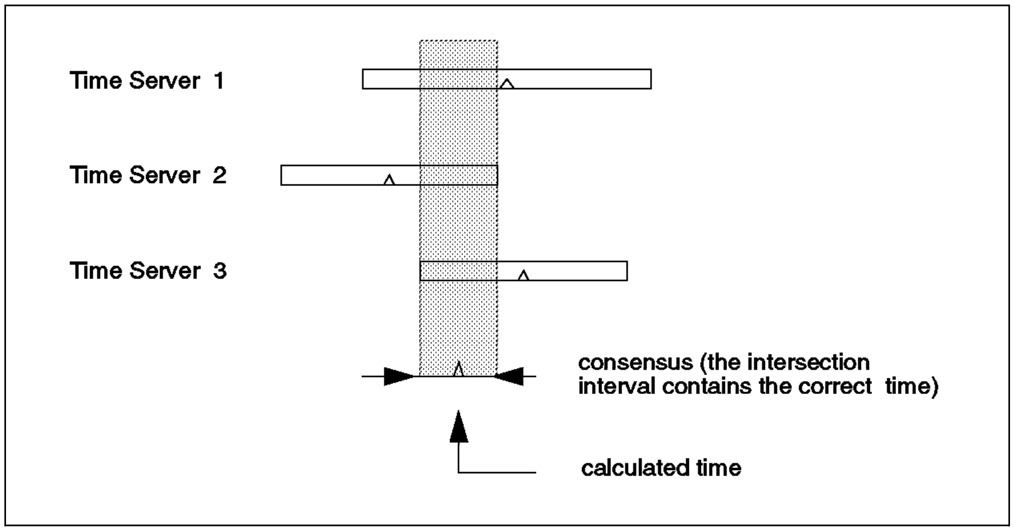

Time Server 1
Time Server 2
Time Server 3
consensus (the intersection interval contains the correct time)
calculated time

Figure 29. Time Intervals

The DTS daemon first adjusts the intervals to take into account communication and processing delays. It then builds the intersection of the intervals. The intersection builds the new inaccuracy interval and its midpoint yields the computed time.

One or more of the servers might be totally wrong about the time of day. In this case, the intersection of the intervals received by the client might be null. If one time interval does not intersect with the majority, it is ignored and that server is marked faulty. The faulty server will not be used anymore to obtain time values until the list is purged.

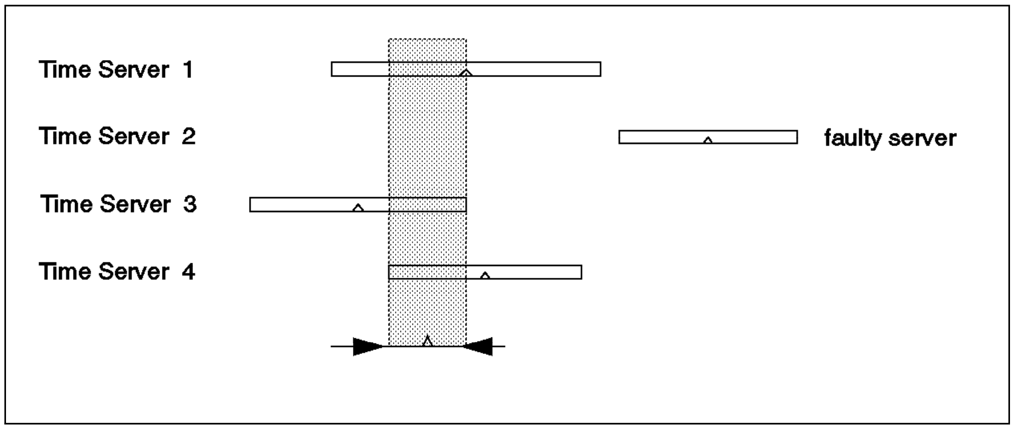


Figure 30. Faulty Server

Figure 30 illustrates how a faulty server is detected. The DTS daemon was configured to require four time servers. Server 2 delivers a time interval that does not intersect with the others and is marked faulty.

4.3.3 Adjusting the Clock

When any DCE node running a DTS daemon has detected a difference between its local time (time on the machine where the daemon is running) and the correct time it has calculated from time values obtained from DTS servers, it usually adjusts its own clock. Under normal circumstances, such adjustment should never cause a clock to move backwards to provide for proper event sequencing.

The update can be gradual or abrupt. Usually, however, it is desirable to update the clock gradually. Instead of changing the system time, we adjust the number of clock ticks that make up one second. The tick increment is modified until the correct time is reached. In other words, if a clock is normally incremented 10 milliseconds at each clock interrupt, and the clock is behind, then the clock register will instead be incremented 10.1 milliseconds at each clock tick until it shows the correct time. By default, we adjust the clocks by one percent for a period long enough to effect the desired change. So, in fact, we slow down or speed up the system clock.

Note: DCE on AIX directly manipulates the system clock in the way described above. OS/2 does not allow changes to the increment value. It changes the clock abruptly and uses a separate DCE clock register that is continuously adjusted. As a consequence, standard OS/2 commands which display the system clock′s value may, at times, display another time than the DTS commands.

If a clock is discovered to be wrong, 30 days for example, adjusting the time gradually would take a long time. If a clock is off by more than a given value (the *Error Tolerance* in DCE 1.0.x or the *tolerance* in DCE 1.1), then it is set immediately to the correct time rather than adjusted gradually. The *tolerance* now defaults to five minutes. It used to be ten minutes in DCE 1.0.x.

4.3.4 Notion of Epochs

The DTS works with epochs. Before one server can obtain time values from another, the servers must have the same epoch number. Epochs divide the DTS implementation into logically separate areas; servers only synchronize with other servers that have the same epoch numbers (clients do not have epoch numbers).

To allow a new notion of time to be propagated throughout a cell, times are interchanged along with an epoch identifier (a simple integer).

4.3.5 Manually Setting a Correct Time Within a Cell

To introduce a new correct time within a cell, a new epoch is created (by incrementing the epoch number). The machine that has received the new time then ignores times passed to it by other servers and vice versa.

An administrator then changes the epoch of the other DTS servers in the cell. When a server is moved to another epoch, it does not advertise the time until it has received the current time within the new epoch from some other server. For more information on how to do a manual adjustment, see 4.6.2, "Setting the System Time" on page 94 in the DTS Administration section.

4.4 DTS Time Format

The next section will explain the DTS time format.

4.4.1 Universal Time Coordinated (UTC)

The Universal Time Coordinated (UTC) is, by international agreement, based on atomic clocks. However, also by international agreement, UTC is occasionally adjusted to the time observed at the Greenwich meridian. This adjustment is performed by addition (or subtraction) of *leap seconds*, which may be added to (or subtracted from) UTC at the end of any month.

Since it is not known sufficiently far in advance when these *leap seconds* will occur, the DCE Distributed Time Service does not have built-in knowledge about when to apply *leap seconds*.

However, since a *leap second* might occur at the end of any month (and since the clock uses UTC), we must add one second to the inaccuracy of our clocks at the end of each month. The added inaccuracy induced by a possible *leap second* will be dealt with by some external, knowledgeable source.

4.4.2 Time Zones or Time Differential Factor (TDF)

The system clocks on all systems in the world store the same time value in their clock registers, in theory. It is UTC. Environment variables in AIX and OS/2 make sure that time zones and seasonal changes are taken into consideration when time is displayed or set. Users always see or enter their local time, and the commands that read or manipulate the system clocks add or subtract the local time difference to the UTC stored in the system clock register.

For example, in Austin, which is in the Central Standard Time zone, the environment variable CST6CDT is used in AIX. AIX also knows the rules for seasonal changes. It applies the changes from/to Daylight Savings Time automatically. In the summer, we are five hours behind UTC; in the winter, we are six hours behind.

DCE uses a Time Differential Factor (TDF) component in their absolute time values to account for time zones and seasonal changes. The time and date displayed are local times, and they are derived from UTC stored in the system clock register and the TDF.

The TDF factor is correctly set up during the initial configuration of the DTS entity. The TDF is also correctly maintained when the time is changed from/to Daylight Savings Time. However, for operations on the system time, DTS uses the TDF and ignores the environment variable. So, when setting an absolute time, the TDF has to be specified.

4.4.3 Time Representation

DTS uses opaque binary timestamps that represent UTC for all of its internal processes. These timestamps cannot be read or disassembled. The DTS API allows applications to convert and manipulate them, but they cannot be displayed. DTS API calls convert timestamps in ASCII text strings, which can be displayed.

4.4.3.1 Absolute Time Representation

An absolute time is a point on a time scale. DTS uses the UTC time scale. Absolute time measurements are derived from system clocks or external time-providers. DTS records the time in an opaque binary timestamp that also includes the inaccuracy and other information.

When an absolute time is displayed, DTS converts the time to ASCII text as shown in the following display :

```
dcecp> clock show
    1995-05-24-17:05:21.343-05:00I000.082
```

DTS displays all time values in a format that complies with the International Standards Organization (ISO).

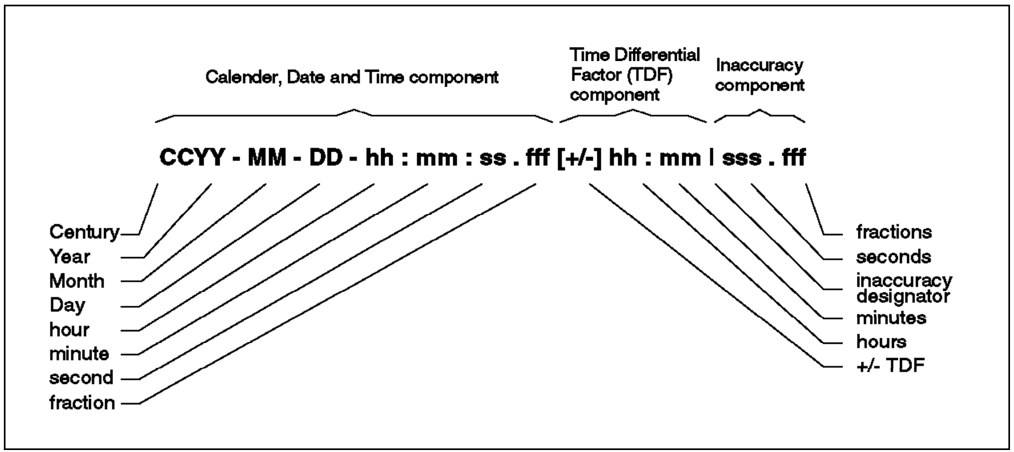

Calender, Date and Time component
Time Differential Factor (TDF) component
Inaccuracy component
CCYY - MM - DD - hh : mm : ss . fff [+/-] hh : mm I sss . fff
Century
Year
Month
Day
hour
minute
second
fraction
fractions
seconds
inaccuracy designator
minutes
hours
+/- TDF

Figure 31. Absolute Time

DTS also allows variation to the ISO format. For more information, consult the *DCE for AIX Application Development Guide - Core Services* (article *Basic DTS Concepts* or use the search argument *ISO-Compliant Time Format Variation*).

Note: The inaccuracy portion of the time is not defined in the ISO standard; so, time that does not contain an inaccuracy is accepted.

Examples of Time Formats	
Represents	**Time Format**
1995-7-14-18:30:00	July 14, 1995 18:30 GMT without inaccuracy (default)
1995-7-14-12:01:00-05:00I100	Local time of 12:01 (17:01 GMT) on July 14, 1995 with a TDF of five hours and an inaccuracy of 100 seconds
12:00	12:00 GMT in the current day, month, and year, with unspecified inaccuracy
1995-7-14	July 14, 1995 00,00 GMT with unspecified inaccuracy

4.4.3.2 Relative Time Representation

A relative time is a discrete time interval that is added to or subtracted from another time. A relative time is normally used as input for commands or system routines.

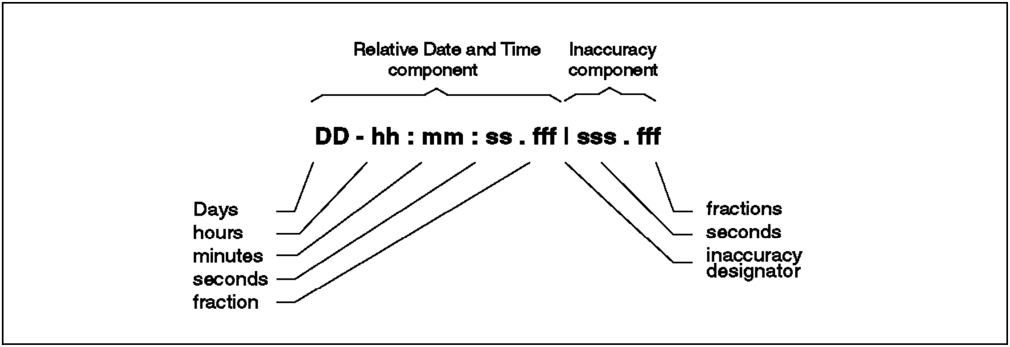


Figure 32. Relative Time Structure

Examples of Relative Time Formats	
Represents	**Time Format**
21-10:30:15.000I000.300	A relative time of 21 days, 10 hours and 30 minutes,15 seconds with inaccuracy of 0.300 seconds
-20.2	A negative relative time of 20.2 seconds with unspecified inaccuracy (default)
10:15.1I4	A relative time of 10 minutes, 15.1 seconds with an inaccuracy of four seconds

4.4.4 Time Structures

The DTS API uses four structures to modify binary times for applications:

- utc
- tm
- timespec
- reltimespec

4.4.4.1 utc Structure

DTS uses 128-bit binary numbers to represent time values internally. These binary numbers, representing time values, are referred to as binary timestamps.

```
typedef struct utc {
idl_byte char_array[16];
} utc_t;
```

The opaque *utc* structure displayed above contains the following information:

- Count of 100-nanosecond units since 00:00:00:00, 15 October 1582 (date of the Gregorian reform to the Christian calendar)
- Count of 100-nanosecond units of accuracy applied to the preceding item
- Time Differential Factor (TDF) expressed as the signed quantity
- DTS version number

The API provides the necessary routines for converting between opaque binary timestamps and character strings that can be displayed and read by users.

4.4.4.2 tm Structure

The *tm* structure is based on the time in years, months, days, hours, minutes, and seconds since 00:00:00 GMT, 1 January 1900.

This structure is defined in the *time.h* header file.

```
  strut tm {
     int tm_sec;      /* Seconds (0-59)            */
     int tm_min;      /* Minutes (0-59)            */
     int tm_hour;     /* Hours   (0-23)            */
     int tm_mday;     /* Day of month  (1-31)      */
     int tm_mon;      /* Month  (0-11)             */
     int tm_year;     /* Year    -1900             */
     int tm_wday;     /* Day of week  (Sunday=0)   */
     int tm_yday;     /* Day of year  (0-364)      */
     int tm_isdst;    /* non zero if Daylight Saving Time */
  }                   /* is in effect                     */
```

4.4.4.3 timespec Structure

The *timespec* structure is normally used in combination with, or in place of, the *tm* structure to provide finer resolution for binary times. The *timespec* structure provides the number of seconds and nanoseconds since the base time of 00:00:00 GMT, 1 January 1970.

This structure is in the *dce/utc.h* header file.

```
  tyedef struct timespec {
    time_t tv_sec; /* seconds since of 00:00:00 GMT, 1 January 1970 */
    long tv_nsec;   /* additional nanoseconds since tv_sec */

  } timespec_t;
```

4.4.4.4 reltimespec Structure

The *reltimespec* structure represents relative time. This structure is similar to the *timespec* structure, except that the first field is signed in the *reltimespec* structure.

This structure is in the *dce/utc.h* header file.

```
tyedef struct reltimespec {
  time_t tv_sec;    /* Seconds of relative time */
  long tv_nsec;   /* Additional nanoseconds of relative time */
} reltimespec_t;
```

4.5 External Time Providers

To prevent the whole cell from drifting away from the UTC time, the notion of correct time should come from an outside source. An atomic clock or some other time service, such as the Internet NTP (Network Time Protocol), might be used to feed DTS.

There are several devices that can acquire the UTC time values via radio, satellite or telephone. These devices can provide standardized time values to computer systems. DTS servers can synchronize with time providers by means of the time provider interface (TPI). The TPI is a DCE interface with an IDL file that specifies the communications between the DTS server process and the time provider process. The time provider process is user-written. It has to read and interpret the external time value and listens to RPC calls from the DTS server. See 4.5.1, "Time Provider Interface (TPI)" below for more details on the TPI.

If an external time provider process is available that answers a DTS server′s calls to the TPI interface, that time server synchronizes *only* with the TP. If the calls are not served via the TPI, the DTS server synchronizes with other DTS servers. This more accurate time value with a usually small interval is picked up by the other DTS entities through their regular synchronizations, thus keeping the synchronized time in the cell close to UTC.

Let only one time service manage the time

To prevent deadlocks or loops, let only one time service adjust the systems′ clocks in the DCE cell. If an NTP server is used as an external time provider and is running on the same machine as the DTS server, use the ***null_provider*** time provider process. It tells DTS, via the TPI, not to adjust the clock (because NTP has already done it) and only sets an inaccuracy value.

If you use NTP or another time service to synchronize the time on your systems in the DCE cell, do not configure DTS and vice versa.

You can run time providers on multiple systems in the cell. The DTS servers on these machines then only synchronize via the TPI and provide their time values to the other DTS entities for their synchronization process.

4.5.1 Time Provider Interface (TPI)

The external time provider is implemented as an independent process that communicates with a DTS server process through Remote Procedure Call (RPCs).

- DTS daemon is the client
- Time-Provider process (TP process) is the server

Both the RPC client (DTS daemon) and the server (TP process) must be running on the same system. If connected to a TP, a time server uses this value to set its own clock. By assigning a very small confidentiality interval, this time value gets a high weight in the time calculation procedure of other DTS daemons.

The RPC interface used by the DTS daemon is predefined and consists of two procedures. The interface is called the Time Provider Interface (TPI). The TP process has to implement the code executing these procedures. The TP process is user-written. It reads the time provided by an external device or another time provider, transforms it into UTC time, assigns an inaccuracy value, and sends time values back when called by the DTS daemon. Example programs for several time providers are supplied for AIX in /usr/lpp/dce/examples/dts. They are ready to be compiled and used. OS/2 does not provide any time provider examples. The two procedures defined by the TPI are:

- ***ContactProvider()***

  This procedure is the first called by DTS. It verifies that the TP process is running and obtains a control message that DTS uses for subsequent communications with the TP process and for synchronization after it receives the timestamps. As part of the answer to this call, the TP process tells the DTS daemon whether it is allowed to change the system clock to the transmitted time values. If another time provider, such as an NTP server, updates the system clock, then the DTS daemon should not also do it.

- ***ServerRequestProviderTime()***

  After the TP process is successfully contacted, this procedure is called by DTS to obtain the timestamps from the external time-provider. The timestamps contain an inaccuracy value, which is usually small. This gives this DTS server a high weight in the synchronization process of other DTS entities.

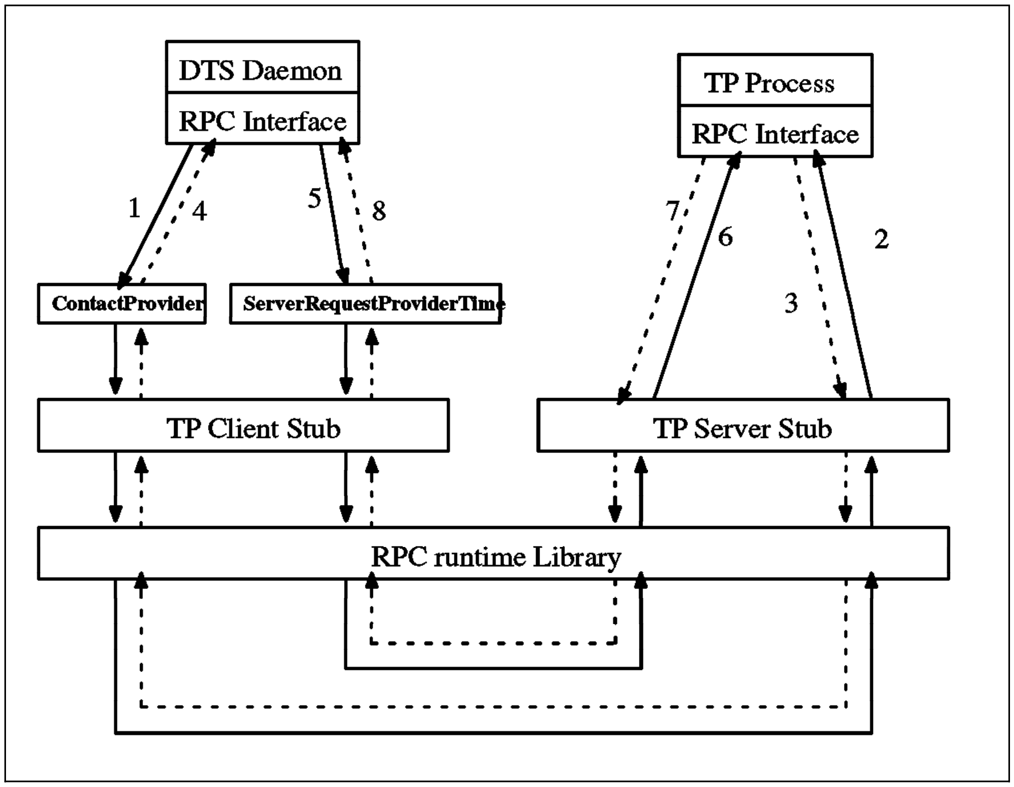


Figure 33. DTS Daemon and TP Process RPC Calling Sequence

Steps 1 through 4 in Figure 33 show the execution of *ContactProvider()* procedure that checks the existence of an external time provider. Steps 5 through 8 show the flow of the *ServerRequesterProviderTime()* procedure used to obtain time values with an inaccuracy interval.

For more information on the TPI, see the *DCE for AIX Application Development Guide - Core Services*.

4.6 DTS Administration

To manage DTS, you can use the DCE Control Program, dcecp, which is new in DCE 1.1. The previous dtscp used in DCE 1.0.x is still available. The advantage of dcecp over dtscp is that you can manage DTS daemons of remote systems without having to log into the remote system.

In the following examples, we will always show the new command first and then the dtscp that achieves the same result.

4.6.1 Show the Time

To show the DTS time of the local machine, use the following command:

```
# dcecp
dcecp> clock show
1995-05-25-14:59:58.527-05:00I-----
dcecp> quit
#
# dtscp
dtscp> show current time
Current Time              = 1995-05-25-15:01:11.711-05:00I-----
dtscp> quit
```

With the following command, you will display all attribute values for the DTS entity on the local node:

```
dcecp> dts show
{tolerance +0-00:05:00.000I-----}
{tdf -0-05:00:00.000I-----}
{maxinaccuracy +0-00:00:00.100I-----}
{minservers 1}
{queryattempts 3}
{localtimeout +0-00:00:05.000I-----}
{globaltimeout +0-00:00:15.000I-----}
{syncinterval +0-00:10:00.000I-----}
{type clerk}
{clockadjrate 10000000 nsec/sec}
{maxdriftrate 50000 nsec/sec}
{clockresolution 10000000 nsec}
{version V1.0.1}
{timerep V1.0.0}
{autotdfchange no}
{nexttdfchange 1995-10-29-01:00:00.000-06:00I0.000}
{status enabled}
{localservers
 {name /.../itsc7.austin.ibm.com/hosts/ev7/self}
 {timelastpolled 1995-08-03-15:58:43.422-05:00I-----}
 {lastobstime 1995-08-03-15:58:48.803-05:00I-----}
 {lastobsskew +0-00:00:05.381I-----}
 {inlastsync TRUE}
 {transport RPC}}
```

```
dtscp> show all
Error Tolerance          = +0-00:05:00.000I-----
Local Time Differential Factor = -0-05:00:00.000I-----
Maximum Inaccuracy       = +0-00:00:00.100I-----
Servers Required         = 1
Query Attempts           = 3
Local Set Timeout        = +0-00:00:05.000I-----
Global Set Timeout       = +0-00:00:15.000I-----
Synchronization Hold Down = +0-00:10:00.000I-----
Type                     = Clerk
Clock Adjustment Rate    = 10000000  nsec/sec
Maximum Clock Drift Rate  = 50000  nsec/sec
Clock Resolution         = 10000000  nsec
DTS Version              = V1.0.1
Time Representation Version = V1.0.0
Automatic TDF Change     = FALSE
Next TDF Change          = 1995-10-29-01:00:00.000-06:00I0.000
```

4.6.2 Setting the System Time

DTS only lets you set the time on DTS servers. The time can either be changed monotonically or abruptly. A monotonic change is continuously done by slowing down or speeding up the clock adjustment rate as described above in 4.3.3, "Adjusting the Clock" on page 86. To initiate a monotonic change on DTS server, ev7, issue the following commands from any system:

```
dcecp> clock show /.:/hosts/ev7/dts-entity
1995-05-25-14:58:19.651-05:00I5.119
dcecp> clock set -to 1995-05-25-15:02:00-05.00I01.00 /.:/hosts/ev7/dts-entity
dcecp> clock show /.:/hosts/ev7/dts-entity
1995-05-25-15:02:11.877-05:00I2.151

dtscp> update time 1995-05-25-15:02:00-05.00I01.00
```

The new time and inaccuracy (one second in the above example) you specify must form a smaller interval than the current system interval (5.119 seconds) and be contained in the current interval. In other words, the new time must be more accurate. To manually set a new time, you should have access to a trusted time reference. Type the command in, and wait until the time reference reaches the desired time; then press <enter> to execute the command.

Don′t forget the TDF

When you set a new time, remember that DTS is dealing with local time. So, don′t forget to specify the time with the correct TDF (-05.00 for Austin during Daylight Savings Time). Otherwise, when you omit the TDF, your local time is set to UTC and, because the TDF is still in effect, it is actually set back five hours. The DTS commands do not honor the environment variables used by the operating system, they only use the TDF.

If the system clock is way out of synchronization and the problem that caused the error has been fixed, you might want to change the system time abruptly. Also, if the new time falls out the time interval of the other DTS servers, DTS only lets you perform an abrupt change. Such a change can only be done together with a change of the epoch. This prevents this server from being declared faulty. To initiate an abrupt change in combination with an epoch change on the local DTS server, issue the following command:

```
dcecp> clock set -to 1995-05-25-15:10:00-05.00I01.00 -abruptly -epoch 1
dcecp> clock show
1995-05-25-15:10:11.567-05:00I1.011

dtscp> change epoch 1 time 1995-05-25-15:10:00-05.00I01.00
```

Note: The AIX date command to change the time or date also changes the DTS time. So, be careful; don′t use the date command to change the time in a DCE environment.

Beware of abrupt changes

DCE servers, especially CDS, are largely dependent upon timestamps. An abrupt change, particularly backwards, could mess up the whole DCE cell. For example, when time is adjusted backwards and changes were made to CDS while the time was wrong, CDS does not perform some operations on objects with timestamps specifying future times.

The actions to be performed on a system with a faulty clock depend on the machine′s role in the DCE cell. If it is, for example, a CDS server with master replicas, you might need to install the latest backup of its CDS database. If it is a client, it might be necessary to reconfigure that client only.

On client systems, you can force a synchronization if you need to adjust the system clock outside of the regular synchronization process.

```
dcecp> dts synchronize [-abruptly]
dtscp> synchronize [set clock true]
```

The option for an abrupt change makes the DTS entity perform an immediate clock adjustment to the value resulting from the synchronization.

4.6.3 Changing Roles of a Time Servers

You can change a Local Time Server to a Global Time Server with one of the following commands:

```
dcecp> dts configure -global
dtscp> advertise
```

To remove the server from the /.:/cell-profile and make it local-only:

```
dcecp> dts configure -nonglobal
dtscp> unadvertise
```

To modify a time server to a courier role, use one of the two following commands:

```
dcecp> dts modify -change {courierrole courier}
dtscp> set courier role courier
```

To reconfigure the time server as non-courier server, use one of the two following commands:

```
dcecp> dts modify -change {courierrole noncourier}
dtscp> set courier role noncourier
```

With the dcecp subcommand, dts show, you can see the change.

4.7 Platform-Specific Implementation

This section gives a summary of the platform-specific implementation differences for the DTS.

4.7.1 Distributed Time Service on AIX

AIX implements DTS servers and clerks. It provides example implementations for external time providers that are ready to be compiled and run.

The DTS directly affects the system clock. A monotonic adjustment of the system clock executed by DTS is also observed with the AIX `date` command. On the other hand, a change of time and date with the AIX `date` command are like an abrupt change for DTS.

4.7.1.1 Routines with Unsupported Features

In the `utc_mkreltime` DTS routine, on the input argument, *inacctm*, the `tm_zone` structure is not ignored.

4.7.1.2 Removed Command

The `dtss-graph` command, which converts a DTS synchronization trace record into Postscript in DCE 1.0.3 on AIX 3.2.5, is not supported anymore.

4.7.2 Distributed Time Service on OS/2 Warp

OS/2 implements DTS servers and clerks. There is no sample code for external time providers.

The DTS has a separate clock register which is gradually adjusted upon monotonic changes. The system clock is always abruptly changed. For a period of time, when DTS is performing a gradual adjustment, the time returned by OS/2 commands and DTS commands may be different.

4.7.3 Distributed Time Service on DOS Windows

The DCE Distributed Time Service API is not supported in OSF DCE for DOS Windows. Only DTS clerks, and no DTS servers, are implemented on this platform.

Chapter 5. Distributed File Service

The Distributed File Service (DFS) is a DCE application that provides global file sharing. Access to files located anywhere in interconnected DCE cells is transparent to the user. To the user, it appears as if the files were located on a local drive. DFS servers and clients may be heterogeneous computers running different operating systems.

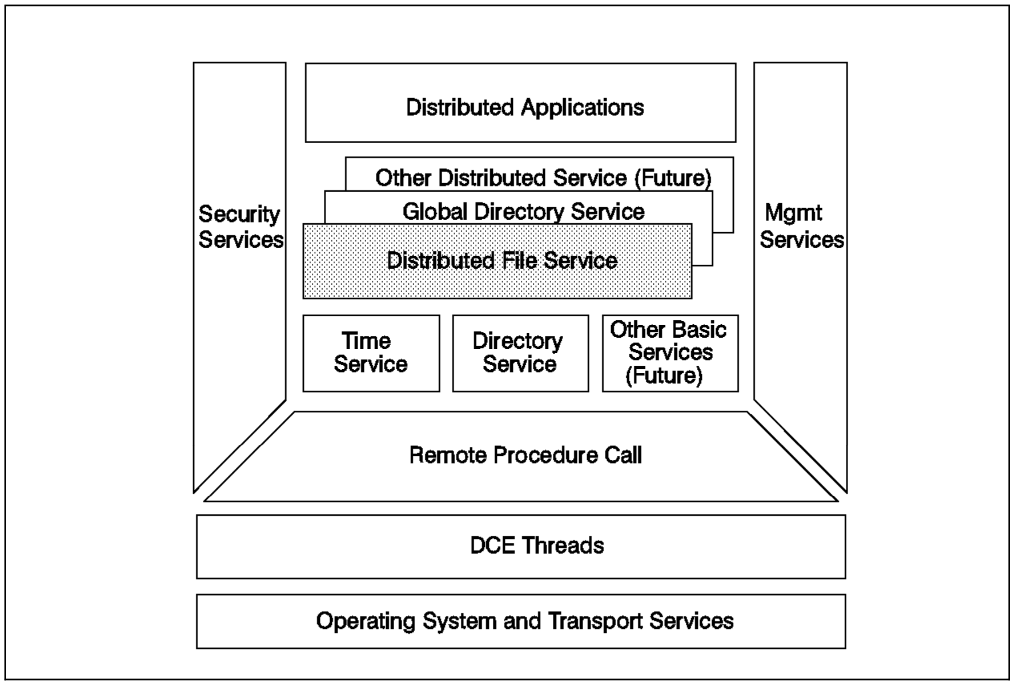


Figure 34. OSF DCE Distributed File Service

The origin of DFS is Transarc Corporation′s implementation of the Andrew File System (AFS) from Carnegie-Mellon University. DFS conforms to POSIX 1003.1 for file system semantics and POSIX 1003.6 for access control security. DFS is built onto and integrated with all of the other DCE services and was developed to address identified distributed file system needs, such as:

- Location transparency
- Uniform naming
- Good performance
- Security
- High availability
- File consistency control
- NFS interoperability

Administrators must understand the concepts and components of the DFS. By creating a useful fileset hierarchy, default ACLs, replication sites, and backup concepts, they can make the DFS use easy for their end-users. Users do not have to know anything about DFS except the pathnames of DFS files; they can access these files just like local files. Maybe advanced users will manage their own ACL definitions to limit access permissions on their files.

This chapter gives a very brief overview of the DFS concepts. You can find more details, as well as installation and administration instructions, in the *The*

Distributed File System (DFS) for AIX/6000 and the *Using and Administering AIX DCE 1.3* redbooks or in the *DCE for AIX DFS Administration Guide and Reference*.

5.1 DFS Architecture

DFS follows the client/server model, and it extends the concept of DCE cells by providing DFS ***administrative domains***, which are an administratively independent collection of DFS server and client systems within a DCE cell.

There may be many DFS file servers in a cell. Each DFS file server runs the ***file exporter*** service which makes files available to DFS clients. The file exporter is also known as the *protocol exporter*.

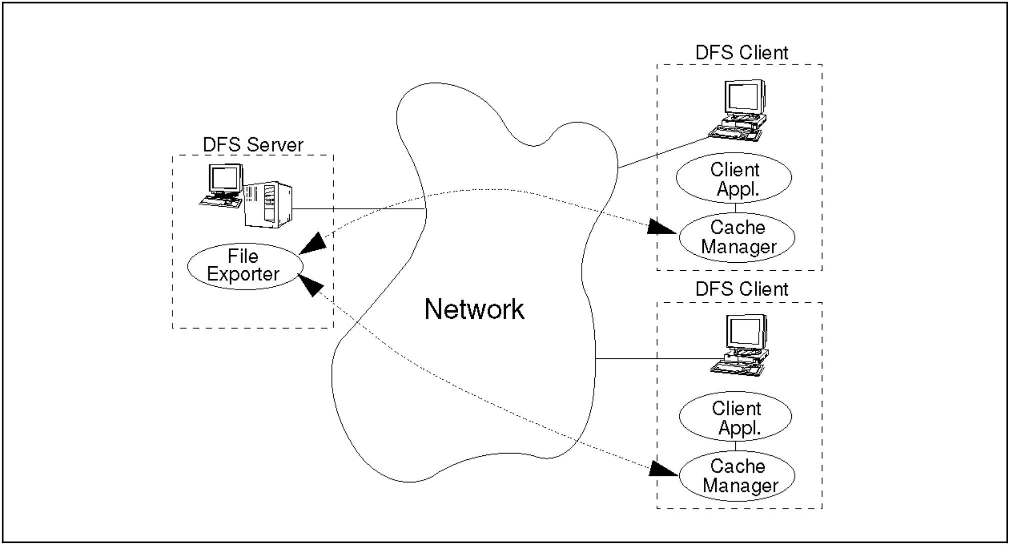


Figure 35. Architecture of the Distributed File Service (DFS)

DFS clients run the ***cache manager***, an intermediary between applications that requests files from DFS servers. The cache manager translates file requests into RPCs to the file exporter on the file server system and stores (caches) file data on disk or in memory to minimize server accesses. It also ensures that the client always has an up-to-date copy of a file.

5.2 DFS File Server

The DFS file server can serve two different types of file systems:

- Local File System (LFS), also known as the Episode File System
- Some other file system, such as the UNIX File System (UFS)

Full DFS functionality is only available with LFS and includes:

- High performance
- Log-based, fast restarting filesets for quick recovery from failure
- High availability with replication, automatic updates and automatic bypassing of failed file server
- Strong security with integration to the DCE security service providing ACL authorization control

Figure 36 on page 99 shows the LFS structure. It uses ***aggregates*** which are physically equivalent to a standard UNIX partition or a logical volume in AIX.

An aggregate holds DFS *structural data* which contains information about the structure and location of data on the aggregate. Each aggregate can host multiple ***filesets***, which is a hierarchical grouping of files managed as a single unit. A fileset corresponds logically to a directory in the file tree. The place at which the fileset is attached to the file tree is known as a ***mount point***. DFS appears as single file tree although the filesets may physically reside on file systems hosted by several computers.

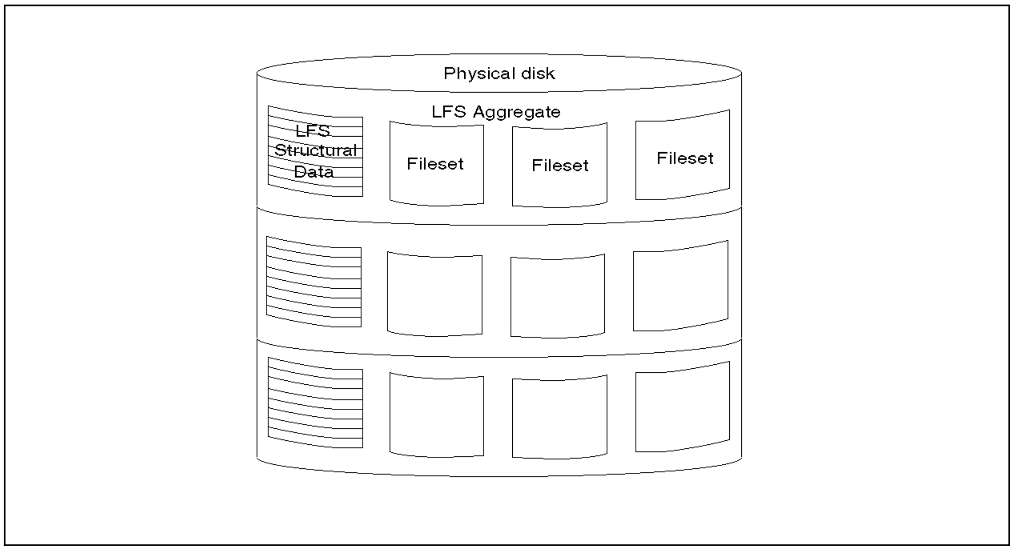


Figure 36. Structure of Local File System (LFS)

The fileset mechanism is used to provide effective support for the advanced features of DFS, for support of replication, and to permit simplified system management of such tasks as reconfiguration and backup. System administrators can easily move LFS filesets from one aggregate to another to ensure availability and to balance the load on file systems. Information on fileset location is saved in the ***fileset location database (FLDB)***. When the DFS clients needs a file, it queries the FLDB which returns the location of a DFS server holding the fileset. If the pathname stretches over several filesets, this look-up process can be an iteration of multiple file server and FLDB accesses the first time a file is looked up.

5.3 File Naming

DFS uses the Cell Directory Service (CDS) name ***/.:/fs*** as a junction to its self-administered namespace. This CDS entry is an RPC group that contains the binding information to the FLDB servers which, as a replicated service, administer the DFS namespace. DFS objects of a cell (files and directories) build a file system tree rooted in /.:/fs of every cell. Since the DFS junction (/.:/fs) can be globally accessed through its full name (/.../cellname/fs), all DFS file systems become part of a global file system.

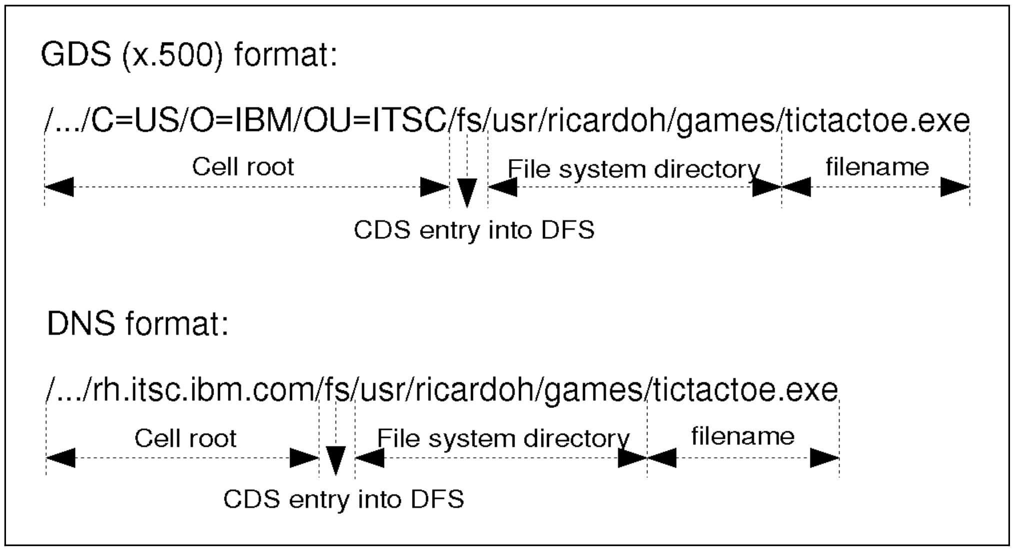


Figure 37. Naming Convention in DFS

Figure 37 shows an example of a file in the global namespace using GDS (X.500) syntax and DNS syntax. The local name can also be abbreviated to /:/usr/ricardoh/games/tictactoe.exe. The /: abbreviation is extended to /.../local_cell/fs.

5.4 Performance

Performance is one of the main goals of DFS, and it achieves it by including features, such as:

- Cache manager — Files requested from the server are stored in cache before being passed to the client so that the client does not need to send requests for data across the network every time the user needs a file. This reduces load on the server file systems and minimizes network traffic, thereby improving performance.
- Multithreaded servers — DFS servers make use of DCE threads support to efficiently handle multiple file requests from the clients.
- RPC pipes — The RPC pipe facility is extensively used to transport large amounts of data efficiently.
- Replication — Replication support allows efficient load-balancing by spreading out the requests for files across multiple servers.

5.5 File Consistency

Using copies of files cached in memory at the client side can cause problems when the file is being used by multiple clients in different locations. There must be file consistency control so that clients can be sure to see other client′s changes and also to have their changes done.

DFS uses a token mechanism to synchronize concurrent file accesses by multiple users. A DFS server has a token manager which manages the tokens that are granted to clients of that file server. On the client side, the cache manager is responsible to comply with the token control. Using this mechanism, DFS ensures that users are always working with the most recent version of a file. The whole process is transparent to the user.

Figure 38 shows the token control done by the DFS server.

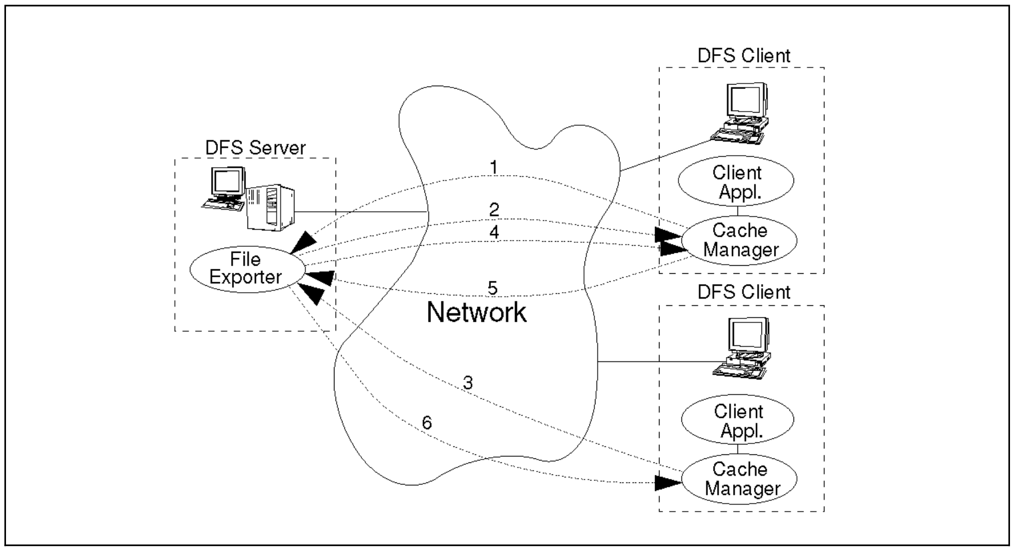


Figure 38. DFS Server's Token Control

1. The client sends a file read request to the DFS server; multiple read tokens can coexist at the same time.
2. If no other client has a write token to the file, DFS sends the data of the file and a read token to the client.
3. The DFS server now receives a file write request from a second client; only one write token can exist at a time and cannot coexist with any read token.
4. The DFS server sees that the file is being used by the first client and sends it a read token revoke.
5. If the first client is not using the file at that time, it accepts the revoke and inform the server.
6. The server now can send a write token to the second client.

All these steps are automatically done and are transparent to the client application. After accepting a revoke, the client discards the file in the cache and gets a new copy from the server the next time the file is needed.

5.6 Availability

Replication of LFS filesets on multiple servers are provided for better availability in DFS. Every fileset has a single read/write version and multiple read-only replicas of that fileset. The read/write version is the only one that contains modifiable versions of directories and files in that fileset; the read-only replicas cannot be modified and can be placed at various sites in the file system. Every change in the read/write fileset is reflected in the replicated filesets.

If there is a crash of a server system housing a replicated fileset, the work is not interrupted, and the client is automatically switched to another replica.

There are more features to improve availability, such as:

- DFS LFS log-based file system — DFS logs information about operations that affect the metadata associated with aggregates and filesets so that it can rapidly return to a consistent state when restarted.
- DFS LFS administration — Allows system administrators to perform routine administration, such as backups and fileset moves between disks, while a server is in operation and available to users. Fileset moves and backups are only possible within a DCE cell. When files need to be moved to another cell, traditional tools need to be used, and ACLs need to be manually transferred.

5.7 Security

DCE security provides DFS with authentication of user identities, verification of user privileges and authorization control. Using the DCE security′s ACL mechanism, DFS extends the UNIX permissions (read, write, execute) by providing precise definitions of access rights for directories and files (owner, insert and delete). Note that only LFS backups provide preservation of ACL information; backups with operating-system commands loose the ACLs.

5.8 Platform-Specific Implementation

Only ***AIX*** provides a full DFS implementation with DFS clients and servers. DFS clients can export all or parts of DFS to NFS; they can act as NFS servers. However, if NFS clients access DFS through such an NFS server, their access is unauthenticated, and their access permissions are very limited. Also, on the AIX platform, there is an ***NFS to DFS Authenticating Gateway*** that allows an NFS user to establish a DCE login on the gateway machine and become an authenticated user. In the latest version of the gateway for AIX Version 4.1 and IBM DCE 2.1, the PC-NFS authentication service (`pcnfsd`) is integrated to allow automatic DCE authentication from a PC-NFS client.

The DCE Beta Version for ***OS/2 Warp*** provides a DFS client.

The current IBM DCE for OS/2 and IBM DCE for DOS Windows do not provide any DFS. The only way for them to access DFS is as an NFS client.

Chapter 6. Installation and Configuration of DCE

This chapter will describe the DCE installation process on AIX and OS/2. We will describe how to create and configure a DCE cell, including client and server components. For DFS configuration, see the redbook *Using and Administering AIX DCE 1.3*.

In the implementation section, we will refer to our DCE test environment, as shown in Figure 39. We will configure *cell1* step by step as shown in Figure 39. The configuration of *cell2* can easily be done following the description for *cell1*. Finally, we describe how to configure intercell communication.

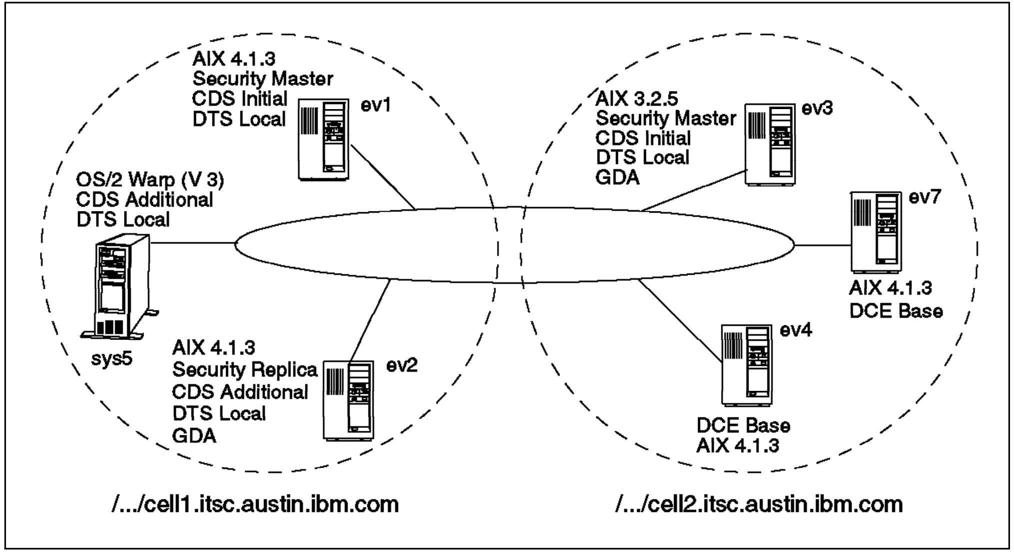


Figure 39. DCE Environment Used in Our Laboratory

There are two phases to the configuration of a DCE cell. During the first phase, or ***initial cell configuration***, certain tasks must be performed to initialize the cell. During the second phase, additional features can be configured into the cell.

The minimum requirements for a cell are:

- One security server
- One CDS server
- At least one DTS server (three per LAN is recommended)

For details about correct DTS layout see Chapter 4, "Distributed Time Service" on page 79. These servers must be configured to initialize a cell. After the cell is up and running, you can configure other components, such as DFS servers, DCE and DFS clients, replicas for the security and CDS servers, a global directory agent, and additional DTS servers or clients.

6.1 AIX Platform

On AIX, you can perform all initial and additional configuration tasks using the standard system management interface tool (SMIT). However, AIX also provides commands to perform these same configuration tasks at the command line, which allows automation by incorporating the commands into shell scripts.

6.1.1 Preparation Steps

If you install DCE for the first time, you should perform some preparation steps before you install the DCE code:

1. Make sure you have at least 100 MB of paging space (`lsps -a` command) and increase it if necessary or add another paging space. For example, to increase an existing paging space by seven partitions (28 MB), run the following command:

   ```
   # chps -s′7′ hd6
   ```

2. Create a separate file system for the /var/dce directory to prevent it from being filled up by AIX subsystems, such as the print spooler and the trace facility. The *DCE V2.1 for AIX: Getting Started* manual (or *Release Notes*) lists the disk space requirements underneath the /var/dce directory, which depends on the components you intend to install. A DCE/DFS client needs 4 MB, and a DCE/DFS server needs something between 24 MB and 50 MB. The following example creates a 24 MB file system:

   ```
   # crfs -v jfs -g rootvg -a size=48000 -m/var/dce -Ayes -prw
   # mount /var/dce
   ```

3. If you run a DFS client on the system, you should also create a separate file system for the DFS cache to isolate it from the rest of the DCE storage. The following example makes room for a 10 MB cache on a 12 MB file system:

   ```
   # crfs -v jfs -g rootvg -a size=24000 -m/var/dce/adm/dfs/cache -Ayes -prw
   # mount /var/dce/adm/dfs/cache
   ```

4. Check the TCP/IP network. Make sure names are correctly resolved using the `host` command and the routing works with the `ping` command.

5. Synchronize the system clocks, for instance, with the following command:

   ```
   # setclock <time_providing_host_name>
   ```

If DCE is already installed, please check whether you comply with the above suggestions, and decide whether you want to change your environment.

6.1.2 Installation

Installation is done using `smit` or the `installp` command. To install DCE call `smit` in the following way:

```
# smitty installp
  →Install/Update From All Available Software
```

Select your input device, and then you′ll get:

```
                   Install/Update From All Available Software

Type or select values in entry fields.
Press Enter AFTER making all desired changes.

                                                        [Entry Fields]
* INPUT device / directory for software               /dev/rmt0.1
* SOFTWARE to install                                []                      +
  PREVIEW only? (install operation will NOT occur)    no                      +
  COMMIT software updates?                            no                      +
  SAVE replaced files?                                yes                     +
  ALTERNATE save directory                           []
  AUTOMATICALLY install requisite software?           yes                     +
  EXTEND file systems if space needed?                yes                     +
  OVERWRITE same or newer versions?                   no                      +
  VERIFY install and check file sizes?                no                      +
  DETAILED output?                                    no                      +



F1=Help             F2=Refresh          F3=Cancel           F4=List
F5=Reset            F6=Command          F7=Edit             F8=Image
F9=Shell            F10=Exit            Enter=Do
```

On the `SOFTWARE to install` line, press **F4**, and select the filesets you want to install on this machine. Set the installation options you want to apply on this machine. Select **Do**. For further general information about installation procedures and SMIT, refer to the *AIX Version 4.1 RISC System/6000 Installation Guide*.

6.1.3 Fast Path

In the following sections for the AIX DCE configuration, we will configure *ev1* as the primary security server, the initial CDS server and a local DTS server. Node *ev2* will be configured as a DCE client first, and then its role will be extended to become a secondary CDS server, a secondary security server, a local DTS server, and a global directory agent (GDA) for intercell communication.

With the following fast path commands, you can achieve the same configuration as described in the rest of this section:

1. On *ev1*, execute the following command:

   ```
   # mkdce -n cell1.itsc.austin.ibm.com sec_srv cds_srv dts_local
   Enter password to be assigned to initial DCE accounts:
   Re-enter password to be assigned to initial DCE accounts:
   ```

2. On *ev2*, execute the following command:

   ```
   # mkdce -R -n cell1.itsc.austin.ibm.com -s ev1 sec_srv cds_second dts_local gda
   Enter password for DCE account cell_admin:
   ```

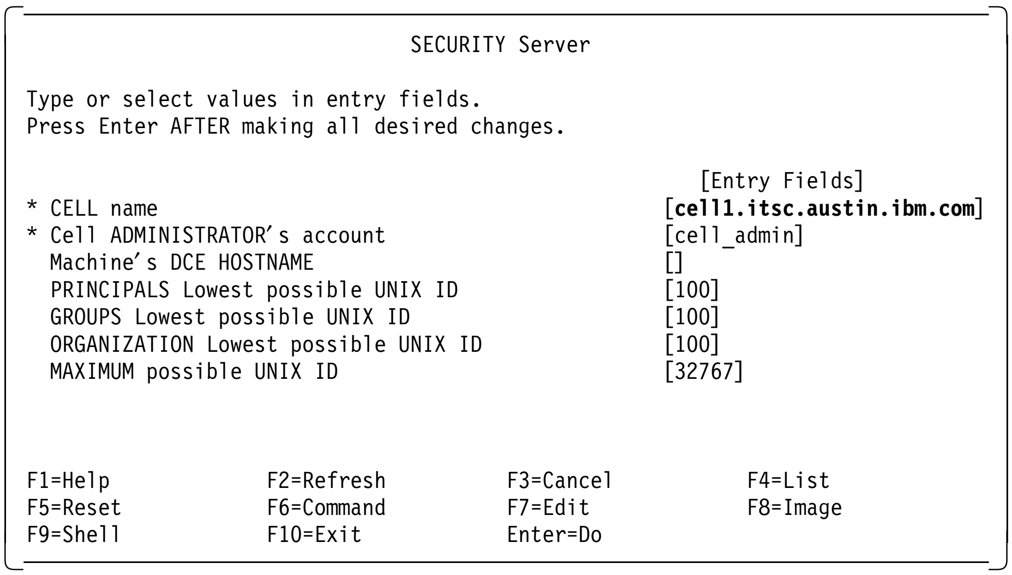
Now you can go directly to the configuration of *sys5*, as illustrated in 6.2, “OS/2 Platform” on page 116.

6.1.4 Configuring the Master Security Server

To configure the master security server for the cell, call smit on *ev1* in the following way:

```
# smitty mkdce
   —Configure DCE/DFS Servers
      —SECURITY Server
```

You will be asked to select whether this server is going to be the primary (master) or secondary (replica) server. Since we are on the initial configuration, you must choose **1 primary**. You will get the following screen:



```
                                  SECURITY Server

Type or select values in entry fields.
Press Enter AFTER making all desired changes.

                                                        [Entry Fields]
* CELL name                                          [cell1.itsc.austin.ibm.com]
* Cell ADMINISTRATOR's account                       [cell_admin]
  Machine's DCE HOSTNAME                             []
  PRINCIPALS Lowest possible UNIX ID                 [100]
  GROUPS Lowest possible UNIX ID                     [100]
  ORGANIZATION Lowest possible UNIX ID               [100]
  MAXIMUM possible UNIX ID                           [32767]



F1=Help             F2=Refresh          F3=Cancel           F4=List
F5=Reset            F6=Command          F7=Edit             F8=Image
F9=Shell            F10=Exit            Enter=Do
```

Fill in the cell name and other information for your cell. You can select a DCE hostname for this machine that is different from the TCP/IP hostname. The default is to take the TCP/IP hostname as the DCE hostname. After filling in the information, select **Do**. When prompted, enter the password for the cell administrator's account. You will get the following messages:

```
Enter password to be assigned to initial DCE accounts:
Re-enter password to be assigned to initial DCE accounts:

Configuring RPC Endpoint Mapper (rpc)...
RPC Endpoint Mapper (rpc) configured successfully

Configuring Security Server (sec_srv)...
Configuring Security Client (sec_cl)...
Security Client (sec_cl) configured successfully

Security Server (sec_srv) configured successfully

Current state of DCE configuration:
rpc          COMPLETE   RPC Endpoint Mapper
sec_cl       COMPLETE   Security Client
sec_srv      COMPLETE   Security Server (Master)
```

The same action could be performed on the command line with the following command:

```
# mkdce -n cell1.itsc.austin.ibm.com sec_srv
```

At this point, the DCE daemon (shown as RPC Endpoint Mapper), the security server and security client are configured on the machine. This command also starts the dced and secd processes.

6.1.5 Configuring the Initial CDS Server

Each cell can only have one primary CDS server. To configure the initial CDS server for a cell, call smit in the following way:

```
# smitty mkdce
  →Configure DCE/DFS Servers
     →CDS (Cell Directory Service) Server
```

Select the **1 initial** option, and you will get the following screen:

```
                      CDS (Cell Directory Service) Server

 Type or select values in entry fields.
 Press Enter AFTER making all desired changes.

                                                      [Entry Fields]
 * CELL name                                         [/.../cell1.itsc.austin>
 * SECURITY Server                                   [ev1]
 * Cell ADMINISTRATOR's account                      [cell_admin]
 * LAN PROFILE                                       [/.:/lan-profile]
   Machine's DCE HOSTNAME                            [ev1]



 F1=Help            F2=Refresh          F3=Cancel           F4=List
 F5=Reset           F6=Command          F7=Edit             F8=Image
 F9=Shell           F10=Exit            Enter=Do
```

If you are configuring the CDS initial server on the same machine where the security server is running, all the information will be filled already. If you were on a different machine, you would have to enter the cell name at the prompt and the host name (or IP address) of the security server. Select **Do**. You will get the following messages:

```
Enter password for DCE account cell_admin:

Configuring Initial CDS Server (cds_srv)...
Configuring CDS Clerk (cds_cl)...
Waiting (up to 2 minutes) for cdsadv to find a CDS server.
Found a CDS server.

        Initializing the namespace ...
           Modifying acls on /.:
           Creating /.:/cell-profile
           .
              << cut some output here
           Modifying acls on /.:/ev1_ch

Initial CDS Server (cds_srv) configured successfully

CDS Clerk (cds_cl) configured successfully

Current state of DCE configuration:
cds_cl       COMPLETE   CDS Clerk
cds_srv      COMPLETE   Initial CDS Server
```

```
rpc          COMPLETE   RPC Endpoint Mapper
sec_cl       COMPLETE   Security Client
sec_srv      COMPLETE   Security Server (Master)
          Press Enter to continue
```

One of the following commands could have achieved the same result:

```
# mkdce -n cell1.itsc.austin.ibm.com -s ev1 cds_srv
# mkdce cds_srv
```

At this point, the RPC, security client, CDS server, and CDS clerk are configured on *ev1*. This command also starts the CDS server (cdsd), the CDS clerk (cds_clerk) and the CDS advertiser (cds_adv) daemons, in addition to the already running daemons dced and secd.

If the CDS Server had been configured on a different machine than *ev1*, you would have to configure a CDS client on *ev1* now. Refer to 6.1.7.1, “Configuring DCE Clients” on page 109 for instructions on how to install a client.

6.1.6 Configuring DTS Servers

You can configure DTS local or global servers. To configure a local DTS server on *ev1*, call smit in the following way:

```
# smitty mkdce
  →Configure DCE/DFS Servers
     →DTS (Distributed Time Service) Server
```

```
                                      DTS Server

 Type or select values in entry fields.
 Press Enter AFTER making all desired changes.

                                                           [Entry Fields]
   Type of SERVER                                        local                 +
   Type of COURIER                                       noncourier            +
 * CELL name                                            [/.../cell1.itsc.austin>
 * SECURITY Server                                      [ev1]
   CDS Server (If in a separate network)                []
 * Cell ADMINISTRATOR's account                         [cell_admin]
 * LAN PROFILE                                          [/.../cell1.itsc.austin>
   Machine's DCE HOSTNAME                               [ev1]



 F1=Help              F2=Refresh          F3=Cancel           F4=List
 F5=Reset             F6=Command          F7=Edit             F8=Image
 F9=Shell             F10=Exit            Enter=Do
```

Select the appropriate Type of SERVER and Type of COURIER for this machine. Select **Do**. The following messages will be displayed:

```
Enter password for DCE account cell_admin:

Configuring Local DTS Server (dts_local)...
Local DTS Server (dts_local) configured successfully

Current state of DCE configuration:
cds_cl       COMPLETE   CDS Clerk
cds_srv      COMPLETE   Initial CDS Server
dts_local    COMPLETE   Local DTS Server
```

```
rpc          COMPLETE   RPC Endpoint Mapper
sec_cl       COMPLETE   Security Client
sec_srv      COMPLETE   Security Server (Master)
          Press Enter to continue
```

It is possible to create the same definition with the following command:

```
# mkdce dts_local
```

If the DTS server is on a different machine than the CDS and Security Server, you will have to configure a DTS entity (server or client) on those machines.

6.1.7 Further Cell Configuration

Once the security, CDS and DTS servers are configured, you have completed the initialization of the cell. Now we can add clients into the cell or configure other servers for replication of the primary servers.

6.1.7.1 Configuring DCE Clients

Configuring clients entails two distinct sets of operations:

- Tasks that require cell administrator authority within the DCE cell.
- Tasks that require *root* user authority on the machine that is to be configured as a DCE client.

Because the cell administrator is unlikely to have root user access to every machine in a cell and because the root users of other machines are unlikely to get cell_admin′s password, these tasks are separated into a split configuration of clients. For configuring security and CDS clients, the configuration can be a two-step process. The cell administrator runs the administrator portion from any machine in the cell to update the namespace and security registry. The root users of the client machines then run the local portion to create necessary files and to start client daemons for all client components. The cell administrator can run the full process on machines where they have root access.

To perform the ***cell administrator′s part of the split configuration method***, call smit on *ev1* in the following way:

```
# smitty mkdce
  →Configure DCE/DFS Clients
     →3 admin only configuration for another machine
```

```
                       Administrator DCE Client Configuration

 Type or select values in entry fields.
 Press Enter AFTER making all desired changes.

                                                       [Entry Fields]
 * CLIENTS to configure                             [sec_cl cds_cl]         +
 * Cell ADMINISTRATOR′s account                     [cell_admin]
   Client Machine DCE HOSTNAME                      [ev2]
 * Client Machine IDENTIFIER                        [ev2]
 * LAN PROFILE                                      [/.../cell1.itsc.austin>



 F1=Help            F2=Refresh          F3=Cancel           F4=List
 F5=Reset           F6=Command          F7=Edit             F8=Image
 F9=Shell           F10=Exit            Enter=Do
```

Select the clients you want to configure on the machine. We show the example for *ev2*. Fill in the TCP/IP hostname (on the `Client Machine IDENTIFIER` prompt) and the DCE hostname (if different from TCP/IP). Select **Do**. You will get the following messages:

```
Enter password for DCE account cell_admin:

Configuring Security Client (sec_cl) for dce_host ev2 on
  machine ev2.itsc.austin.ibm.com ...
Completed admin configuration of Security Client (sec_cl) for
  dce_host ev2 on machine ev2.itsc.austin.ibm.com

Configuring CDS Clerk (cds_cl) for dce_host ev2 on
  machine ev2.itsc.austin.ibm.com ...

           Modifying acls on hosts/ev2
           Modifying acls on hosts/ev2/self
           Modifying acls on hosts/ev2/cds-clerk
           Modifying acls on hosts/ev2/profile
           Modifying acls on /.:/lan-profile

Completed admin configuration of CDS Clerk (cds_cl) for
  dce_host ev2 on machine ev2.itsc.austin.ibm.com

Cell administrator's portion of client configuration has completed
  successfully. Root administrator for ev2.itsc.austin.ibm.com should now
  complete the client configuration on that machine.

          Press Enter to continue
```

You can type the next command instead:

```
# mkdce -o admin -i ev2 -h ev2 sec_cl cds_cl
```

To do the ***local part of the split configuration***, log in as root on the designated machine *ev2*, and call smit in the following way:

```
# smitty mkdce
  →Configure DCE/DFS Clients
     →2 local only configuration for this machine
```

```
                       Local DCE/DFS Client Configuration

 Type or select values in entry fields.
 Press Enter AFTER making all desired changes.

                                                          [Entry Fields]
 * CELL name                                           [cell1.itsc.austin.ibm.>
 * CLIENTS to configure                                [rpc sec_cl cds_cl]     +
 * SECURITY Server                                     [ev1]
   CDS Server (If in a separate network)               []
 * Client Machine DCE HOSTNAME                         [ev2]
      The following fields are used
      ONLY if a DFS client is configured
 * DFS CACHE on disk or memory?                        [disk]                  +
 * DFS cache SIZE (in kilobytes)                       [10000]
 * DFS cache DIRECTORY (if on disk)                    [/var/dce/adm/dfs/cache]



 F1=Help             F2=Refresh          F3=Cancel           F4=List
 F5=Reset            F6=Command          F7=Edit             F8=Image
 F9=Shell            F10=Exit            Enter=Do
```

Fill in the cell name, the clients to configure, the name of security server, and the DCE hostname of the local machine. You will get the following messages:

```
Configuring RPC Endpoint Mapper (rpc)...
RPC Endpoint Mapper (rpc) configured successfully

Configuring Security Client (sec_cl)...
Security Client (sec_cl) configured successfully on the
  local machine

Configuring CDS Clerk (cds_cl)...
CDS Clerk (cds_cl) configured successfully on the
  local machine

Current state of DCE configuration:
cds_cl       COMPLETE   CDS Clerk
rpc          COMPLETE   RPC Endpoint Mapper
sec_cl       COMPLETE   Security Client
          Press Enter to continue
```

Using the ***full configuration method*** on a client system, you can make it in one step if you know the cell administrator's password. On *ev2*, you can call smit with the following path:

```
# smitty mkdce
  —Configure DCE/DFS Clients
     —1 full configuration for this machine
```

```
                       Full DCE/DFS Client Configuration

Type or select values in entry fields.
Press Enter AFTER making all desired changes.

                                                        [Entry Fields]
* CELL name                                          [cell1.itsc.austin.ibm.com>
* CLIENTS to configure                               [rpc sec_cl cds_cl]                +
* SECURITY Server                                    [ev1]
  CDS Server (If in a separate network)              []
* Cell ADMINISTRATOR's account                       [cell_admin]
* LAN PROFILE                                        [/.:/lan-profile]
  Client Machine DCE HOSTNAME                        [ev2]
     The following fields are used
     ONLY if a DFS client is configured
* DFS CACHE on disk or memory?                       [disk]                +
* DFS cache SIZE (in kilobytes)                      [10000]
```

Fill in the required information, and select **Do**.

6.1.7.2 Configuring a Secondary CDS Server

The administrator of the cell may decide to have a secondary cell directory server for backup and availability reasons. When you create a secondary CDS server, the root directory of the CDS is automatically replicated on this server.

To configure a secondary CDS server, call `smit` on *ev2* as follows:

```
# smitty mkdcesrv
  —CDS (Cell Directory Service) Server
     —2 additional
```

```
                      CDS (Cell Directory Service) Server

Type or select values in entry fields.
Press Enter AFTER making all desired changes.

                                                        [Entry Fields]
* CELL name                                          [/.../cell1.itsc.austin>
* SECURITY Server                                    [ev1]
  Initial CDS Server (If in a separate network)      []
* Cell ADMINISTRATOR's account                       [cell_admin]
* LAN PROFILE                                        [/.../cell1.itsc.austin>
  Machine's DCE HOSTNAME                             [ev2]



F1=Help             F2=Refresh          F3=Cancel           F4=List
F5=Reset            F6=Command          F7=Edit             F8=Image
```

All the fields should be automatically filled. Select **Do**. You will be prompted for the cell administrator's password. You will get the following messages:

```
Enter password for DCE account cell_admin:

Configuring Additional CDS Server (cds_second)...

            Modifying acls on ev2_ch

Additional CDS Server (cds_second) configured successfully
```

```
Current state of DCE configuration:
cds_cl       COMPLETE   CDS Clerk
cds_second   COMPLETE   Additional CDS Server
rpc          COMPLETE   RPC Endpoint Mapper
sec_cl       COMPLETE   Security Client
          Press Enter to continue
```

6.1.7.3 Configuring a Security Replica Server

The configuration of a security replica decreases the load on the master security server and keeps the cell functional in case the master security server becomes unavailable. To configure *ev2* as a security replica server, log in as root on *ev2*, and call smit in the following way:

```
# smitty mkdcesrv
  →SECURITY Server
     →2 secondary
```

```
                                SECURITY Server

 Type or select values in entry fields.
 Press Enter AFTER making all desired changes.

                                                        [Entry Fields]
 * CELL name                                          [/.../cell1.itsc.austin>
 * Cell ADMINISTRATOR's account                       [cell_admin]
 * REPLICA name                                       [ev2]
 * SECURITY Server                                    [ev1]
   CDS Server (If in a separate network)              []
 * LAN PROFILE                                        [/.../cell1.itsc.austin>
   Machine's DCE HOSTNAME                             [ev2]



 F1=Help             F2=Refresh          F3=Cancel           F4=List
 F5=Reset            F6=Command          F7=Edit             F8=Image
 F9=Shell            F10=Exit            Enter=Do
```

All fields should be automatically filled. Select **Do**. You will be prompted for the cell administrator's password. You will get the following messages:

```
Enter password for DCE account cell_admin:

Configuring Security Server (sec_srv)...

           Modifying acls on /.:/sec/replist
           Modifying acls on /.:/subsys/dce/sec
           Modifying acls on /.:/sec
           Modifying acls on /.:
           Modifying acls on /.:/cell-profile
Security Server (sec_srv) configured successfully

Current state of DCE configuration:
cds_cl       COMPLETE   CDS Clerk
cds_second   COMPLETE   Additional CDS Server
rpc          COMPLETE   RPC Endpoint Mapper
sec_cl       COMPLETE   Security Client
sec_srv      COMPLETE   Security Server (Replica)
          Press Enter to continue
```

6.1.7.4 Configuring Additional DTS Servers

To configure a machine as a DTS Server, it must be at least a DCE client for all components (except DTS). If the machine is configured as a DTS client, unconfigure it. You must log in as root on the designated machine, and start smit on *ev2* the following way:

```
# smitty mkdcesrv
  →DTS (Distributed Time Service)
```

```
                                    DTS Server

 Type or select values in entry fields.
 Press Enter AFTER making all desired changes.

                                                          [Entry Fields]
   Type of SERVER                                       local               +
   Type of COURIER                                      noncourier          +
 * CELL name                                           [/.../cell1.itsc.austin>
 * SECURITY Server                                     [ev1]
   CDS Server (If in a separate network)               []
 * Cell ADMINISTRATOR's account                        [cell_admin]
 * LAN PROFILE                                         [/.../cell1.itsc.austin>
   Machine's DCE HOSTNAME                              [ev2]



 F1=Help             F2=Refresh          F3=Cancel           F4=List
 F5=Reset            F6=Command          F7=Edit             F8=Image
 F9=Shell            F10=Exit            Enter=Do
```

Select the appropriate type of server and type of courier. Select **Do**. You will be prompted for the cell administrator's password. The following messages will be shown:

```
Enter password for DCE account cell_admin:

Configuring Local DTS Server (dts_local)...
Local DTS Server (dts_local) configured successfully

Current state of DCE configuration:
cds_cl       COMPLETE   CDS Clerk
cds_second   COMPLETE   Additional CDS Server
dts_local    COMPLETE   Local DTS Server
rpc          COMPLETE   RPC Endpoint Mapper
sec_cl       COMPLETE   Security Client
sec_srv      COMPLETE   Security Server (Replica)
          Press Enter to continue
```

6.1.7.5 Configuring the Global Directory Agent

The machine designated to run the GDA should be configured at least as a DCE client. To configure GDA, you must call smit on *ev2* in the following way:

```
# smitty mkdcesrv
  →GDA (Global Directory Agent) Server
```

```
                                  GDA Server

Type or select values in entry fields.
Press Enter AFTER making all desired changes.

                                                       [Entry Fields]
* CELL name                                         [/.../cell1.itsc.austin>
* SECURITY Server                                   [ev1]
  CDS Server (If in a separate network)             []
* Cell ADMINISTRATOR's account                      [cell_admin]
* LAN PROFILE                                       [/.../cell1.itsc.austin>
  Machine's DCE HOSTNAME                            [ev2]




F1=Help             F2=Refresh          F3=Cancel           F4=List
F5=Reset            F6=Command          F7=Edit             F8=Image
F9=Shell            F10=Exit            Enter=Do
```

Because this machine is already a client, all the information should be automatically filled. Select **Do**. When prompted, enter the cell administrator's password. You will get the following messages:

```
Enter password for DCE account cell_admin:

Configuring Global Directory Agent (gda)...
Global Directory Agent (gda) configured successfully

Current state of DCE configuration:
cds_cl       COMPLETE   CDS Clerk
cds_second   COMPLETE   Additional CDS Server
dts_local    COMPLETE   Local DTS Server
gda          COMPLETE   Global Directory Agent
rpc          COMPLETE   RPC Endpoint Mapper
sec_cl       COMPLETE   Security Client
sec_srv      COMPLETE   Security Server (Replica)
           Press Enter to continue
```

6.1.8 Configuring DFS Servers

To configure DFS servers, you have to take several steps:

1. Configure one or more system control machines (SCM server).
2. Configure one or more fileset location database servers (FLDB server).
3. Configure one or more file server machines.

You can create, optionally, one or more fileset replication server machines and one or more backup database machines. For information on how to configure the DFS servers, refer to *Using and Administering AIX DCE 1.3*, GG24-4348, and to *The Distributed File Systems (DFS) for AIX/6000*, GG24-4255.

6.2 OS/2 Platform

On OS/2 you can perform installation and configuration tasks from a graphical user interface. Note that the description of these tasks is based on the IBM DCE for OS/2 Warp Beta 2 code with a release date of August 1995. There may be changes to this in the final product. However, the basic installation and configuration procedures will be similar to those presented here:

1. A series of windows will guide you through the installation. First, you will need to specify the components you want to install on the system, and then the files will be copied from the installation media to the local harddisk. An *OS/2 DCE v2.1* folder will be created containing several program objects for DCE configuration and management.
2. You will be able to start the configuration process by double-clicking a *Configure DCE* object in the *OS/2 DCE v2.1* folder. A series of windows will guide you through the configuration.
3. After the first two steps, DCE is up and running. An administrator can then, for instance, use the object-oriented administration GUI to monitor and/or manage the cell.

Please look for *Release Notes* and/or *README* files which may contain current recommendations, instructions or limitations.

6.2.1 Preparation Steps

The IBM DCE for OS/2 Beta 2 supports TCP/IP and NetBIOS protocols. We use TCP/IP to be able to communicate with the AIX machines. However, you could choose NetBIOS between two OS/2 machines. The DCE protocol sequences for NetBIOS are ncadg_nb_dgram (connectionless) and ncacn_nb_stream (connection-oriented).

Before you install DCE, the network must be properly configured and working:

1. Install MPTS (Multi-Protocol Transport Service) for OS/2 Version 3.0.
2. Configure your network adapter and protocol stacks in MPTS
3. Configure TCP/IP

The MPTS installation is done, for example, from diskettes with the `A:\INSTALL` command. This installs an *MPTS* icon on the desktop. Double-click this icon to configure MPTS.

In the *Configure* window, you need to select the LAPS (LAN adapters and protocols) to configure the network adapter and protocol drivers you use. Be sure to select the TCP/IP protocol. This is done in the *LAPS Configuration* window. Back in the *Configure* window, you must select the Socket MPTS Transport Access. Again, be sure to select TCP/IP. Then, click on **Configure** and eventually on the **Exit** buttons.

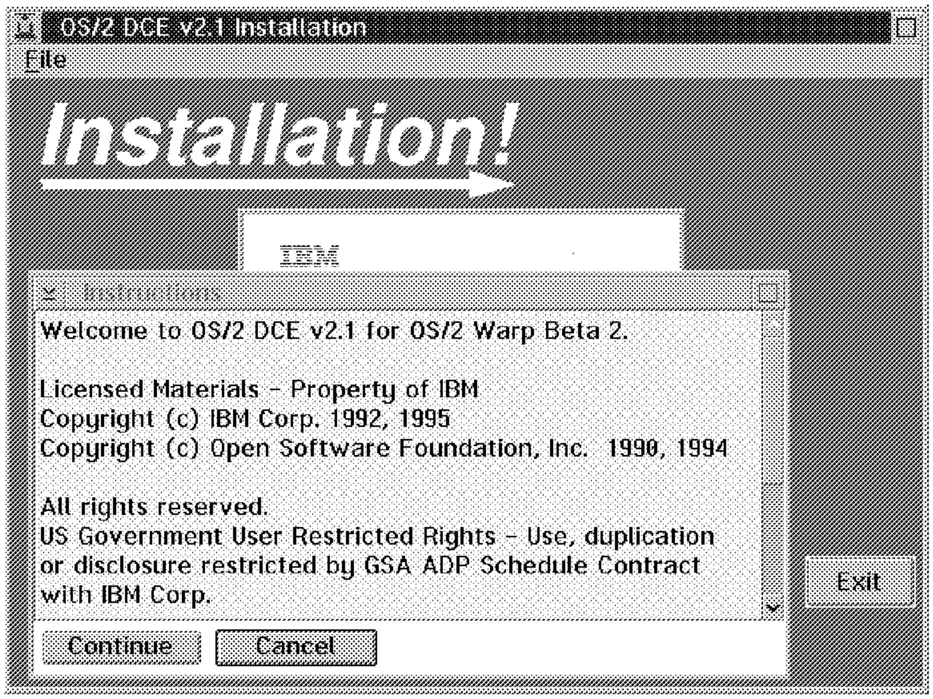
To configure TCP/IP, double-click on *TCP/IP* folder and on the *TCP/IP Configuration* icon within that folder. This brings up the *TCP/IP Configuration* window with a notebook-like interface to fill in the different configuration parameters. On the *Network* page, fill in the IP address and the subnet mask. On the *Routing* page, fill in at least a default route by pushing the **Insert Before** button. On the *Services* page one, fill in the hostname, the domain name and domain name server address. Then, exit the program and test the network, for example, with the `ping` command.

6.2.2 Installation

Assuming Z**:** is the drive that contains the installation software, type the following command at the OS/2 command line:

```
[Z:\] install
```

First, the *OS/2 DCE v2.1 Installation* window is displayed overlapped by an copyright window. This is shown in Figure 40 below.



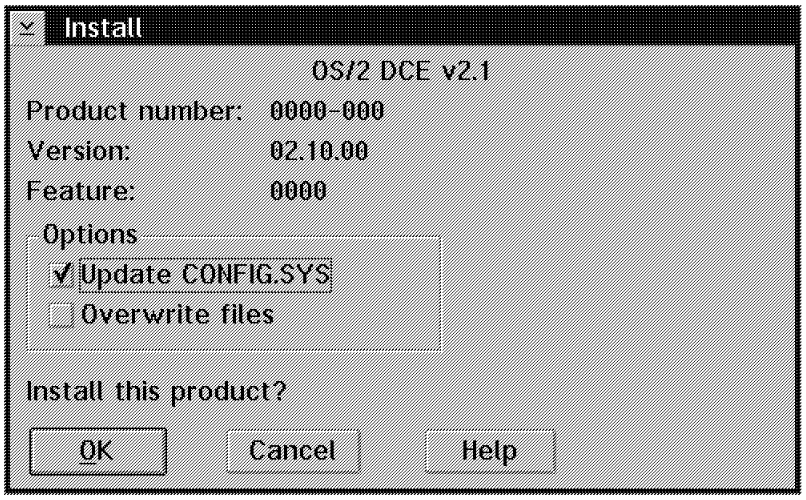
Welcome to OS/2 DCE v2.1 for OS/2 Warp Beta 2.

Licensed Materials - Property of IBM
Copyright (c) IBM Corp. 1992, 1995
Copyright (c) Open Software Foundation, Inc. 1990, 1994

All rights reserved.
US Government User Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Figure 40. DCE Installation Copyright Notice

Push the **Continue** button to get to the *Install* window which is shown in Figure 41 below.



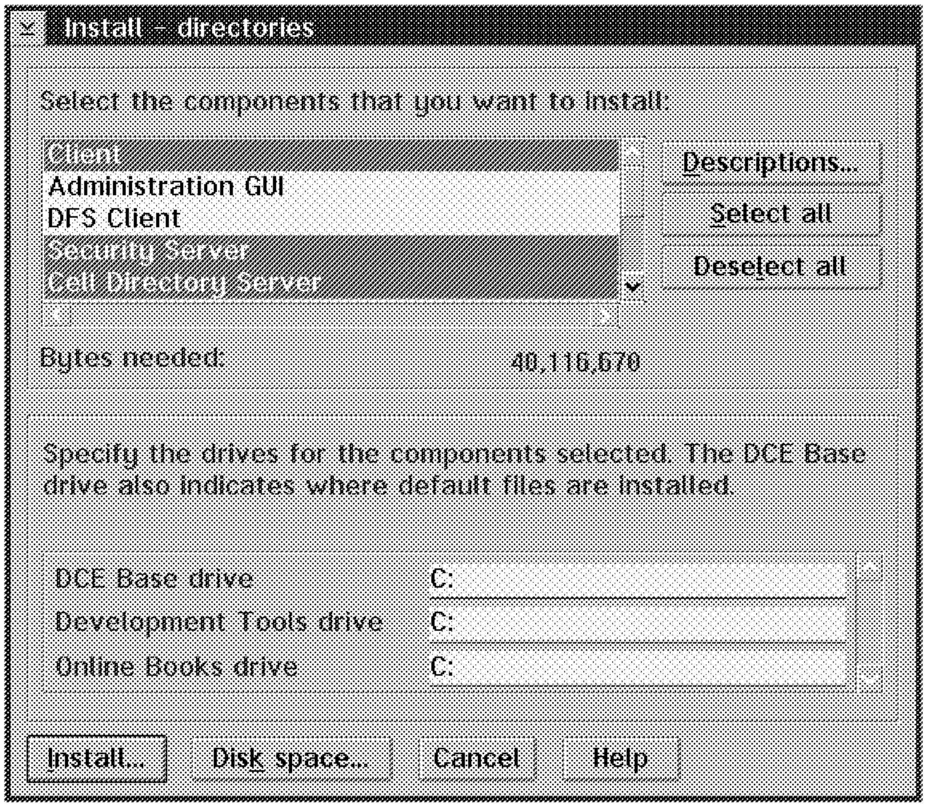
OS/2 DCE v2.1
Product number: 0000-000
Version: 02.10.00
Feature: 0000

[✓] Update CONFIG.SYS
[ ] Overwrite files

Install this product?

Figure 41. DCE Install

The **Update CONFIG.SYS** is already selected. So, click on the **OK** button to confirm this. The *Install — directories* in Figure 42 on page 118 below is displayed.


Install – directories

Select the components that you want to install:

Client
Administration GUI
DFS Client
Security Server
Cell Directory Server

Bytes needed: 40,116,670

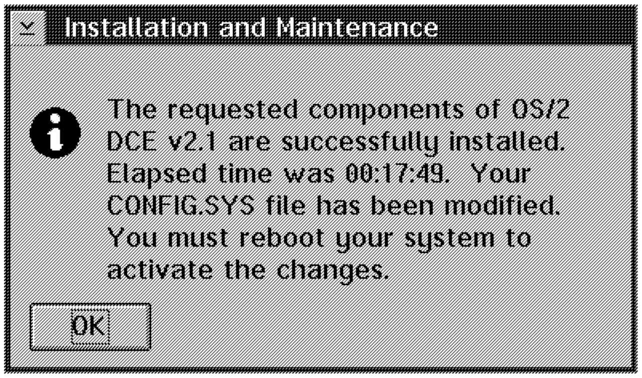
Specify the drives for the components selected. The DCE Base drive also indicates where default files are installed.

DCE Base drive C:
Development Tools drive C:
Online Books drive C:

Figure 42. DCE Component Selection

Select the components you want to install and optionally specify installation path names. This example shows the selection of the DCE Client, Security Server and CDS Server components. In fact, we also selected the DFS components and the Online Documentation for our installation. When you are done, push the **Install** button.

The *Install — progress* window will be displayed. The files are now copied from the installation media into the /opt/dcelocal directory on the target drive. When all files are copied, the *Installation and Maintenance* information window is displayed as shown in Figure 43 on page 119 below.


Installation and Maintenance

The requested components of OS/2 DCE v2.1 are successfully installed. Elapsed time was 00:17:49. Your CONFIG.SYS file has been modified. You must reboot your system to activate the changes.

Figure 43. Installation Complete Notice

When you click on the **OK** button, the *OS/2 DCE v2.1 Installation* window is redisplayed as shown in Figure 44 below.

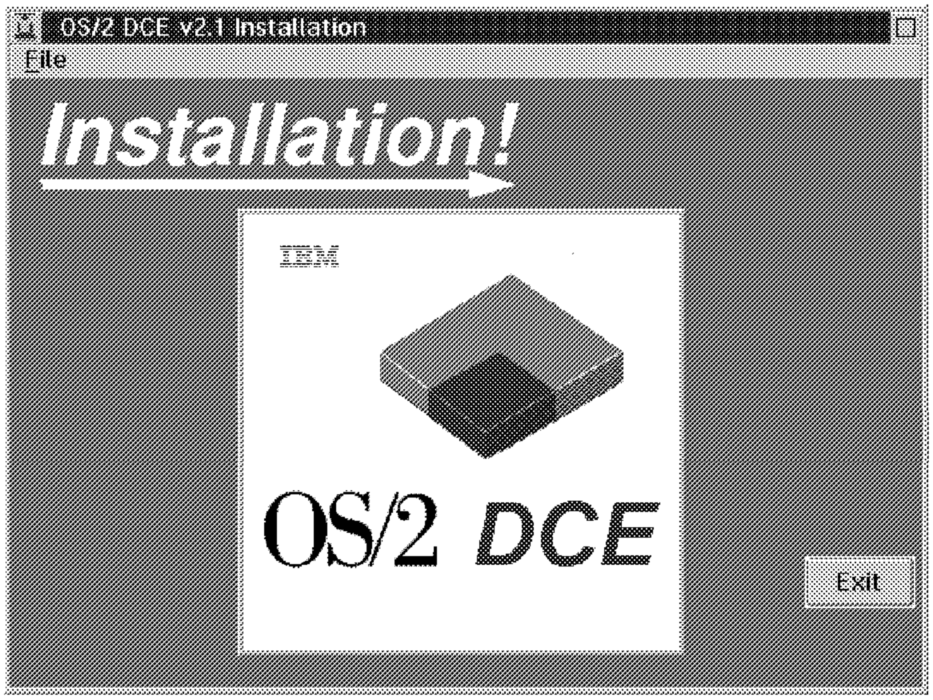


Figure 44. Installation Main Window

Push the **Exit** button, and reboot your machine.

6.2.3 Configuring OS/2 with Master Security and Initial CDS Servers

You can start the configuration process from within the *DCE 2.1 Beta* folder shown in Figure 45 on page 120 by double-clicking on the *DCE Configuration* icon.

Note: In this section, we are going to explain how to configure an OS/2 machine with all components needed for a cell (master security server and initial CDS server). If you want to configure the OS/2 machine as it is shown in our test environment, Figure 39 on page 103, go directly to 6.2.4, “Configuring OS/2 as a DCE Client and an Additional CDS Server” on page 128.

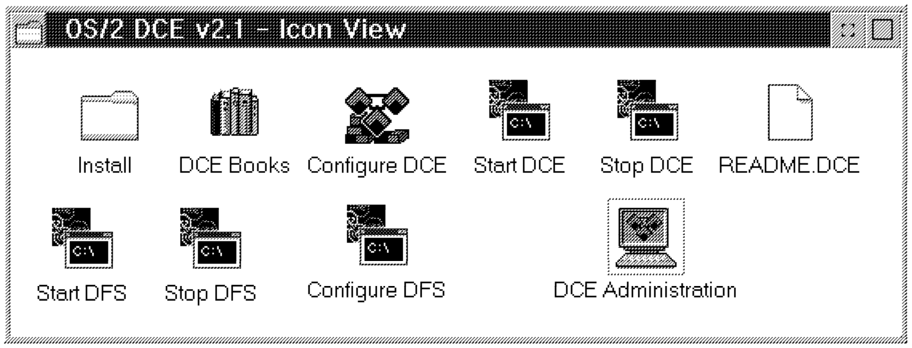
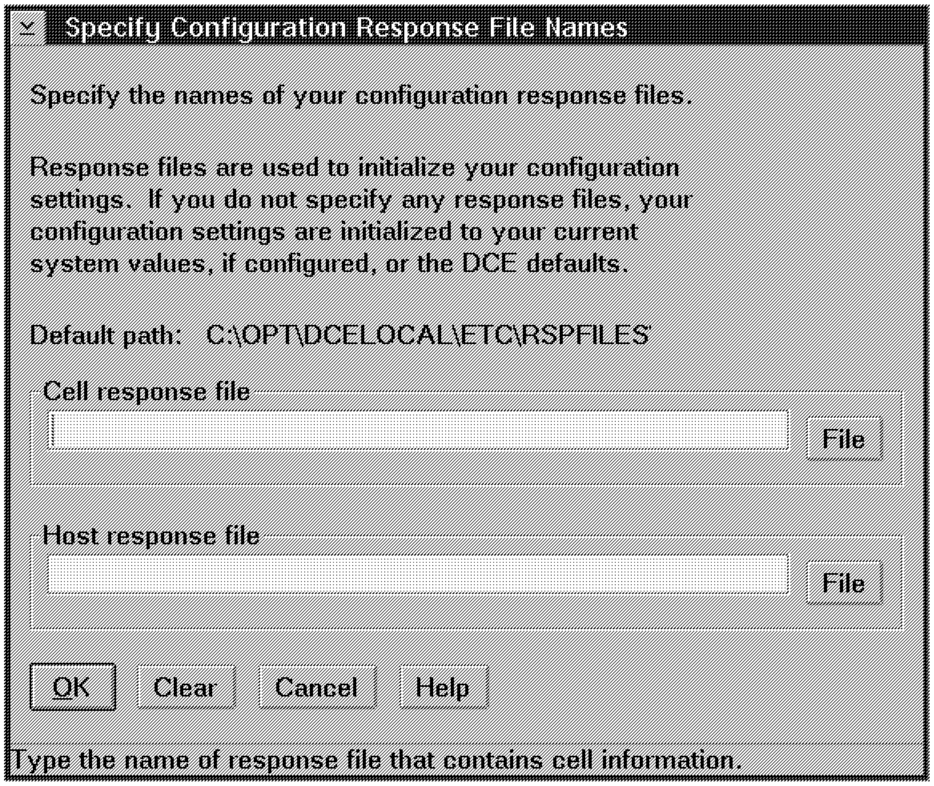


Figure 45. DCE Folder

After starting the configuration process, a welcome window with the IBM logo shows up. Select **Continue**. This will bring up the *Specify Configuration Response File Names* window as shown in Figure 46.



Specify the names of your configuration response files.

Response files are used to initialize your configuration settings. If you do not specify any response files, your configuration settings are initialized to your current system values, if configured, or the DCE defaults.

Default path: C:\OPT\DCELOCAL\ETC\RSPFILES\

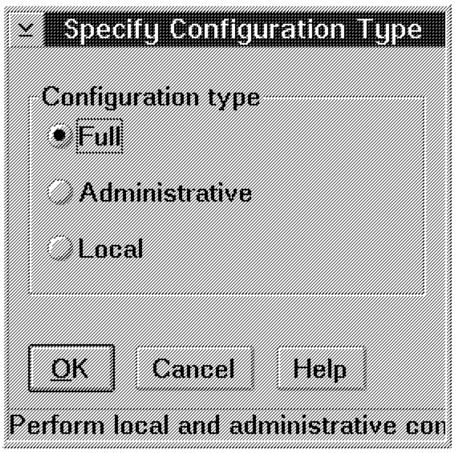
Cell response file

Host response file

Type the name of response file that contains cell information.

Figure 46. Specify Configuration Response File Names Window

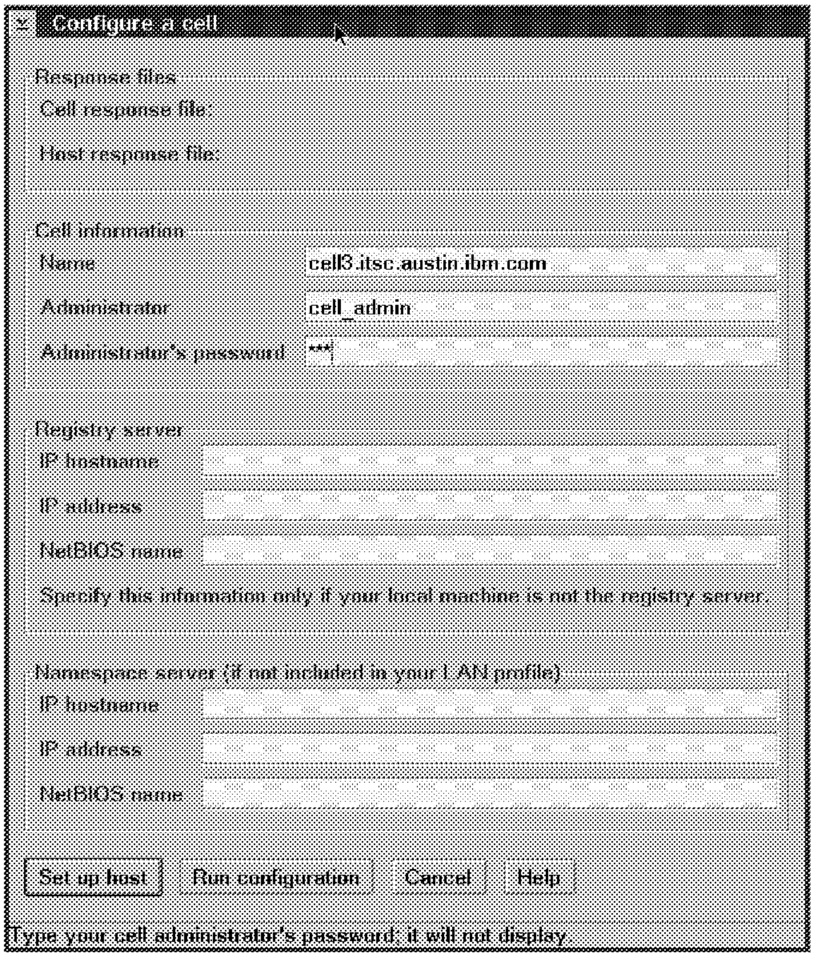
A response file contains the information that you enter in the various window fields during the configuration process. Creating a response file allows you to speed-up and automate the process by eliminating the need to enter the information for each window. Response files are automatically created when you configure DCE. If you have previously created response files, enter their names. Then click on the **OK** button, which will bring up the *Specify Configuration Type* window as shown in Figure 47 on page 121 below.



Perform local and administrative con

Figure 47. Specify Configuration Type Window

You must select the type of configuration you want to perform. This corresponds to the options you have for the installation of a DCE client as explained in 6.1.7.1, “Configuring DCE Clients” on page 109. The **Administrative** option would initiate the administrator part of the split configuration method, whereas the **Local** option stands for the local part of the split configuration. However, since we are configuring an initial cell with all components on this machine, we select **Full**, and push the **OK** button. The *Configure a Cell* window in Figure 48 on page 122 is displayed.


Configure a cell

Response files

Cell response file:

Host response file:

Cell information

Name	cell3.itsc.austin.ibm.com
Administrator	cell_admin
Administrator's password	***

Registry server

IP hostname	
IP address	
NetBIOS name	

Specify this information only if your local machine is not the registry server.

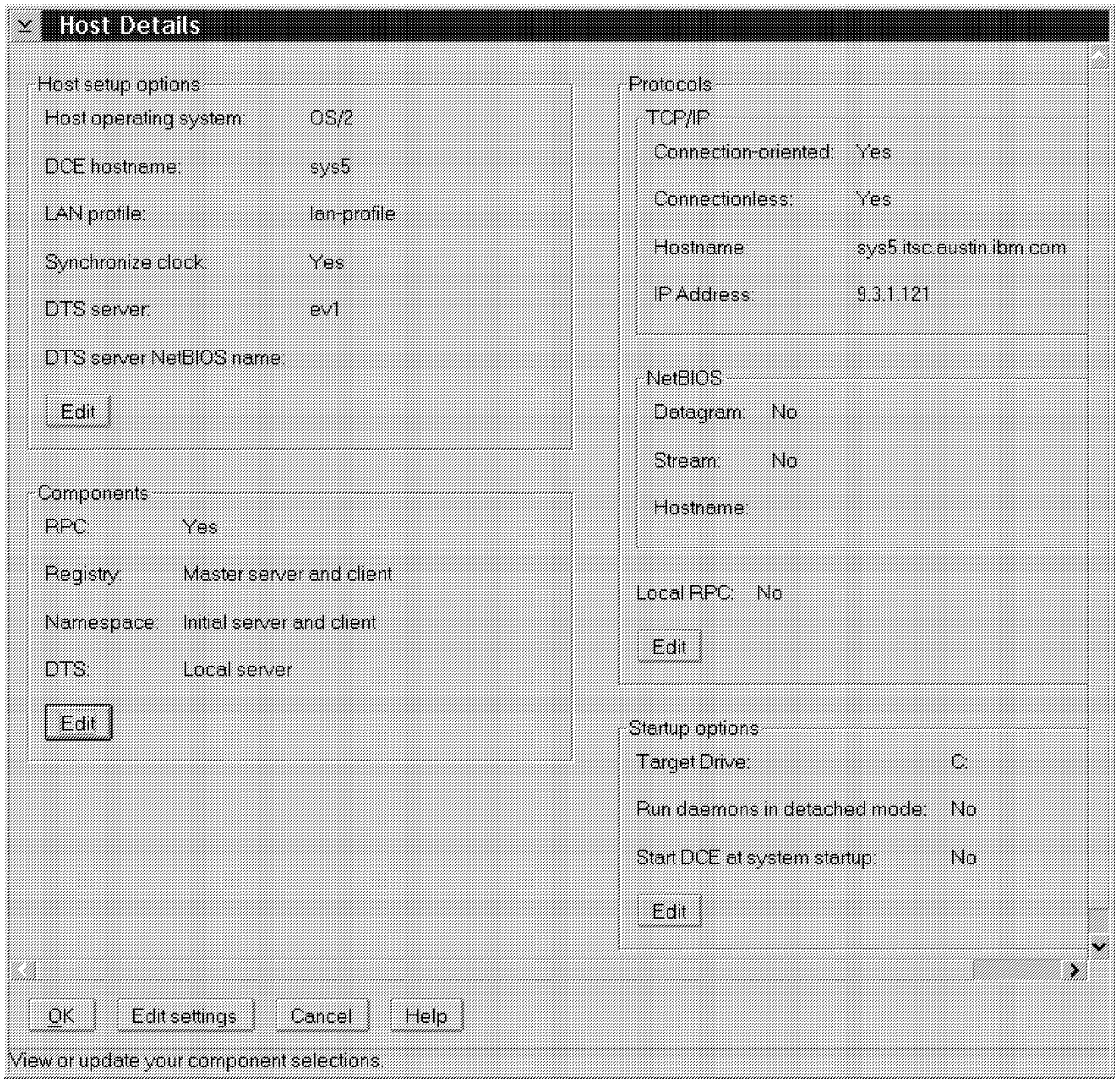
Namespace server (if not included in your LAN profile)

IP hostname	
IP address	
NetBIOS name	

Type your cell administrator's password; it will not display.

Figure 48. Configure Cell Window

Enter the cell name and the cell administrator if you want to overwrite the default of cell_admin and the administrator′s password. Click on the **Set up host** button. The **Host Details** in Figure 49 on page 123 window is displayed.



Host setup options

Host operating system:	OS/2
DCE hostname:	sys5
LAN profile:	lan-profile
Synchronize clock:	Yes
DTS server:	ev1
DTS server NetBIOS name:	

Components

RPC:	Yes
Registry:	Master server and client
Namespace:	Initial server and client
DTS:	Local server

Protocols

TCP/IP

Connection-oriented:	Yes
Connectionless:	Yes
Hostname:	sys5.itsc.austin.ibm.com
IP Address:	9.3.1.121

NetBIOS

Datagram:	No
Stream:	No
Hostname:	

Local RPC: No

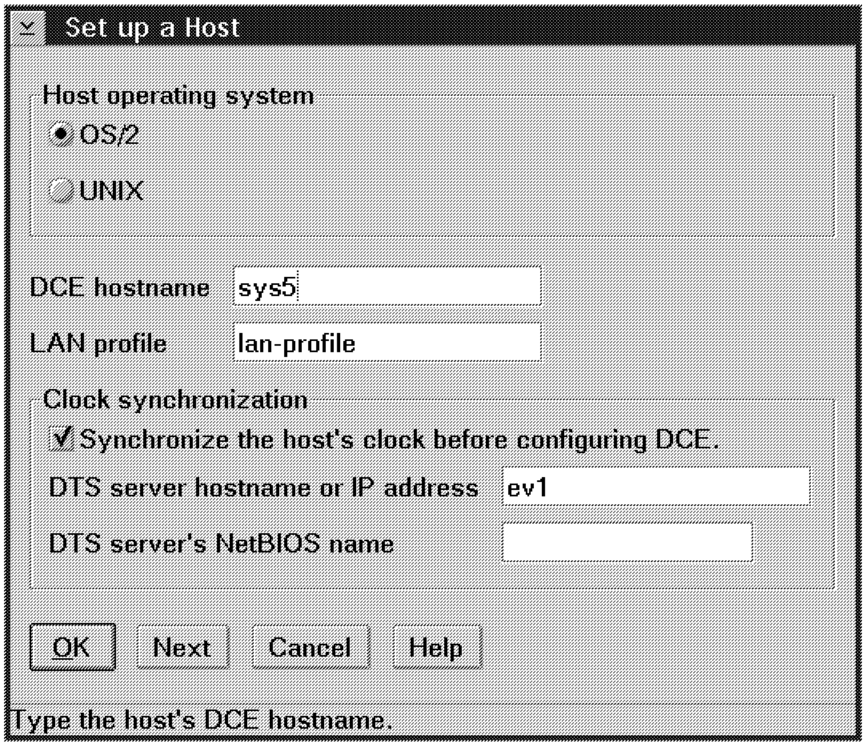
Startup options

Target Drive:	C:
Run daemons in detached mode:	No
Start DCE at system startup:	No

View or update your component selections.

Figure 49. Host Detail Window

The picture shown in Figure 49 already has the values that we are going to configure. To set up the configuration parameters, you can either click on the **Edit** button of a specific options area, or click on the **Edit Settings** to sequentially step through all areas. This is what we are going to do. The first window coming up is the *Set up a Host* window, shown in Figure 50 on page 124 below.


Set up a Host

Host operating system

[✓ OS/2]

UNIX

DCE hostname: sys5

LAN profile: lan-profile

Clock synchronization

[✓] Synchronize the host's clock before configuring DCE.

DTS server hostname or IP address: ev1

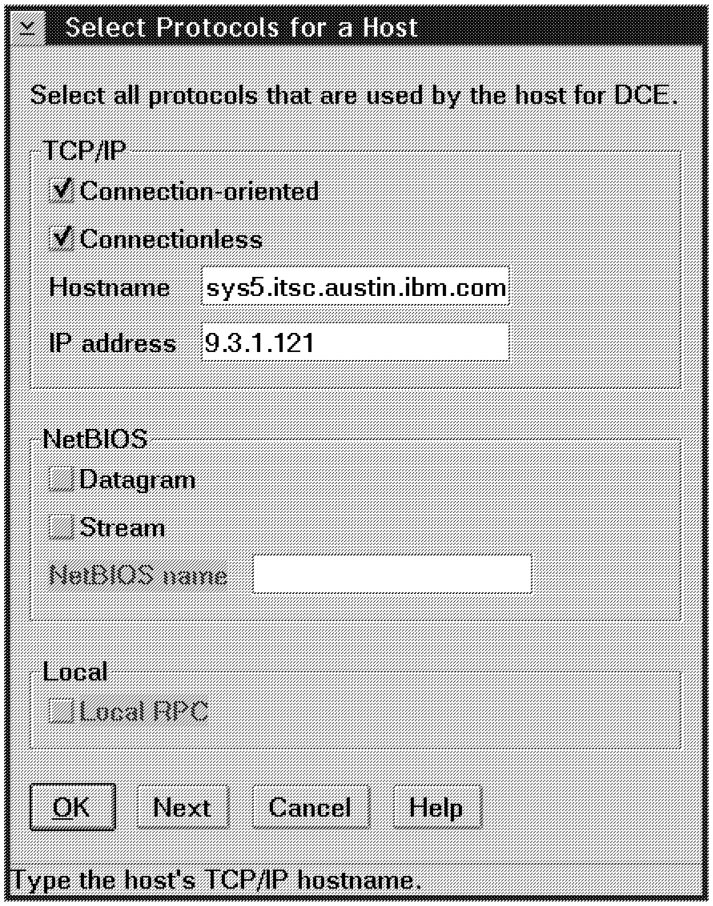
DTS server's NetBIOS name:

Type the host's DCE hostname.

Figure 50. Set Up a Host Window

The **OS/2** operating system is already selected. Enter the DCE hostname of this machine, which is *SYS5*. It is also recommended to synchronize the clock with another node. If you want to do so, select that option, and enter the hostname of a system carrying a good time reference. Then click on the **Next** button.

The *Select Protocols for a Host* window is displayed and shown in Figure 51 on page 125 below.


Select Protocols for a Host

Select all protocols that are used by the host for DCE.

TCP/IP

[✓] Connection-oriented

[✓] Connectionless

Hostname: sys5.itsc.austin.ibm.com

IP address: 9.3.1.121

NetBIOS

[ ] Datagram

[ ] Stream

NetBIOS name:

Local

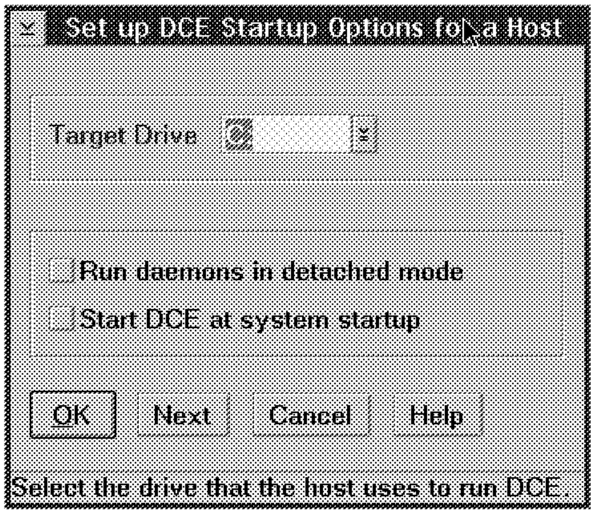
[ ] Local RPC

Type the host's TCP/IP hostname.

Figure 51. Select Protocols for a Host

The TCP/IP configuration is already filled in as it needs to be. Local RPC is not supported in this beta version. Local RPC corresponds to the ncacn_unix_stream RPC protocol and is used in RPC client/server communications within the same machine. It uses faster inter-process communication and bypasses all the network layers.

Then click on the **Next** button. The *Set up DCE Startup Options for a Host* window is displayed, as shown in Figure 52 on page 126.

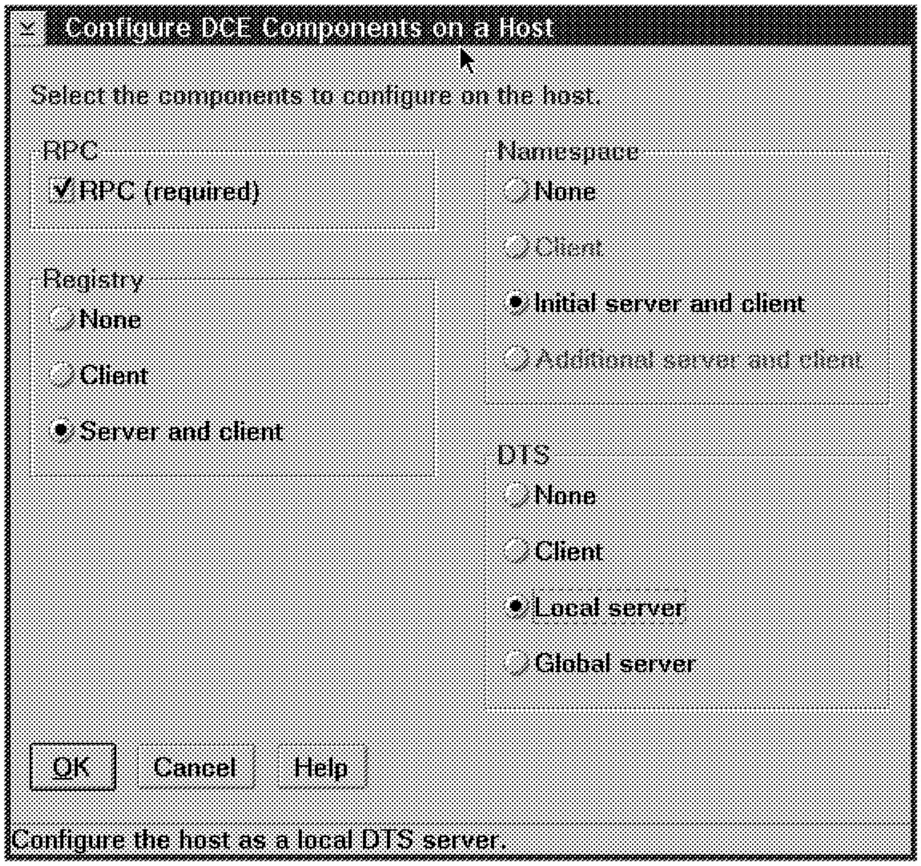


Run daemons in detached mode [ ]
Start DCE at system startup [ ]

Select the drive that the host uses to run DCE.

Figure 52. DCE Start-Up Options for a Host

In this window, you can specify the target drive where the DCE code is located. This is already filled in. You can also choose to automatically start DCE at system startup. Then click on the **Next** button. The *Configure DCE Components on a Host* window is displayed as shown in Figure 53.



Select the components to configure on the host.

RPC: [✓ RPC (required)]

Registry: None, Client, [✓ Server and client]

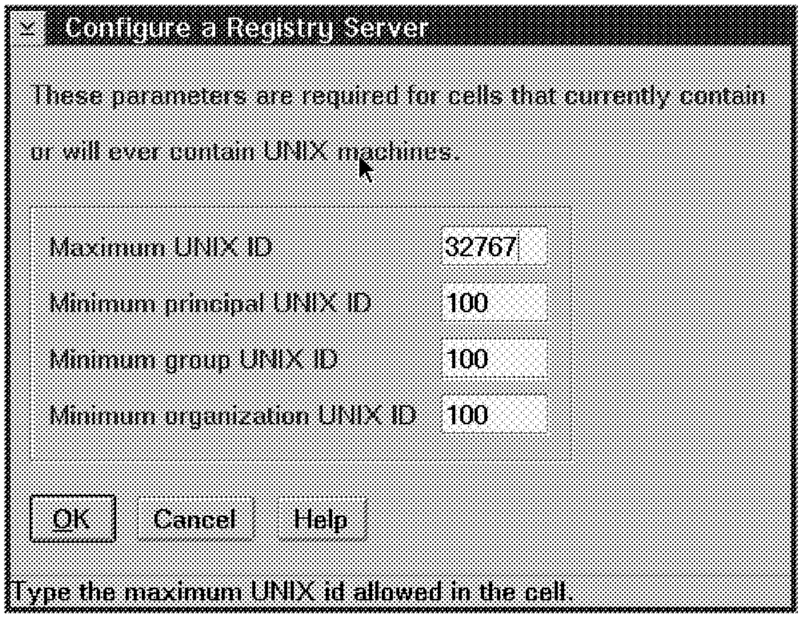
Namespace: None, Client, [✓ Initial server and client], Additional server and client

DTS: None, Client, [✓ Local server], Global server

Configure the host as a local DTS server.

Figure 53. Configure DCE Components on a Host Window

Select the **Server and client** in the Registry options, **Initial server and client** in the Namespace options and **Local server** in the DTS options. The RPC is pre-selected because it is required on every host. Push the **OK** button. Since we have selected to be a security server, the *Configure a Registry Server* window in Figure 54 is displayed.



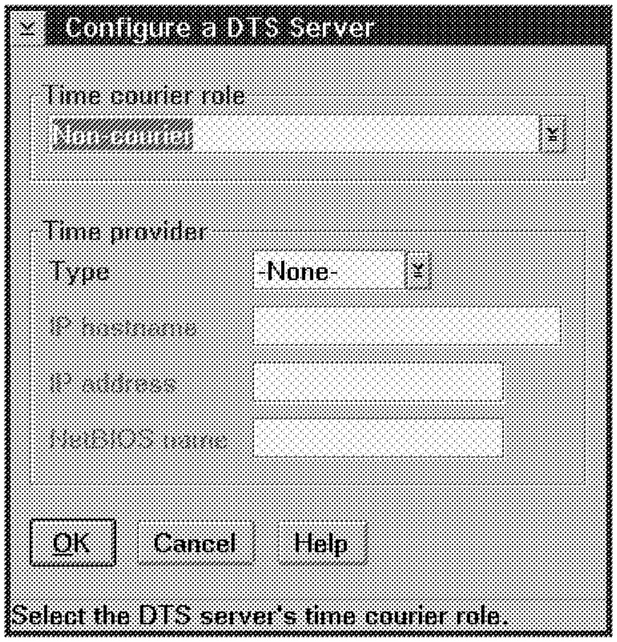
These parameters are required for cells that currently contain or will ever contain UNIX machines.

Maximum UNIX ID	32767
Minimum principal UNIX ID	100
Minimum group UNIX ID	100
Minimum organization UNIX ID	100

Type the maximum UNIX id allowed in the cell.

Figure 54. Configure a Registry Server Window

Click on the **OK** button, unless you want to change any of the UNIX ID bounds. The *Configure a DTS Server* window will be displayed as shown in Figure 55 below.



Time courier role: Non-courier

Time provider

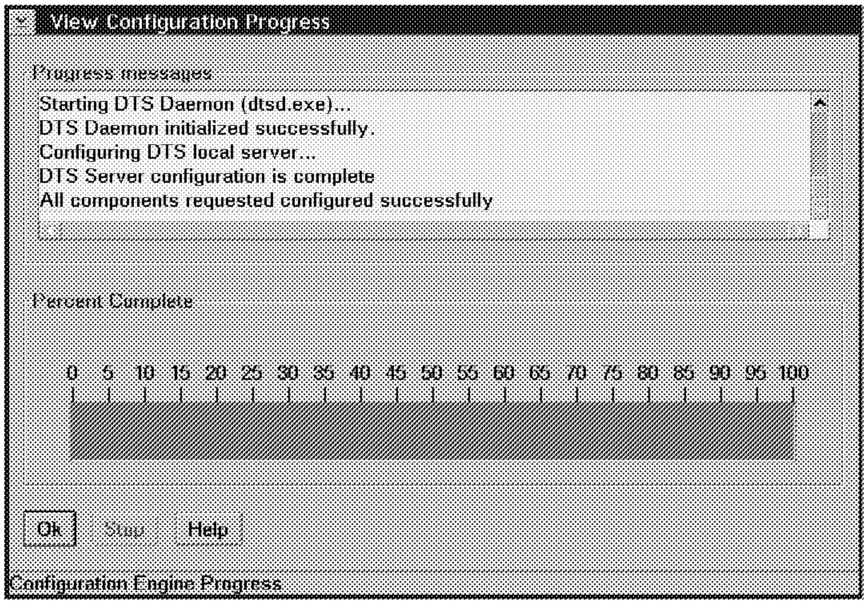
Type	-None-
IP hostname	
IP address	
NetBIOS name	

Select the DTS server's time courier role.

Figure 55. Configure a DTS Server Window

Select the type of courier you want this server to be. If there were an external time provider, you could specify its name here. When you click on the **OK** button, the *Host Details* window as illustrated in Figure 49 on page 123 will be redisplayed. Click on the **OK** button. The *Configure a cell* window is displayed. See Figure 48 on page 122. Now push the **Run Configuration** button. The response files are automatically saved.

The *View Configuration Progress* window is displayed as shown in Figure 56.



```
Starting DTS Daemon (dtsd.exe)...
DTS Daemon initialized successfully.
Configuring DTS local server...
DTS Server configuration is complete
All components requested configured successfully
```

Figure 56. Configuration Progress Window

When configuration is complete, push the **OK** button to return to the *DCE configuration* window and the **Exit** button end the configuration process.

The DCE is already up and running. From a OS/2 command line window, you can run a `dcelogin` and other commands to test the environment.

6.2.4 Configuring OS/2 as a DCE Client and an Additional CDS Server

In the previous section, we configured a one-machine cell with an OS/2 machine as the master security server and initial CDS server. However, for our environment illustrated in Figure 39 on page 103, we want the OS/2 machine to be part of the cell we have already created with AIX machines. In order to add it to our cell, we need to unconfigure it with the following command:

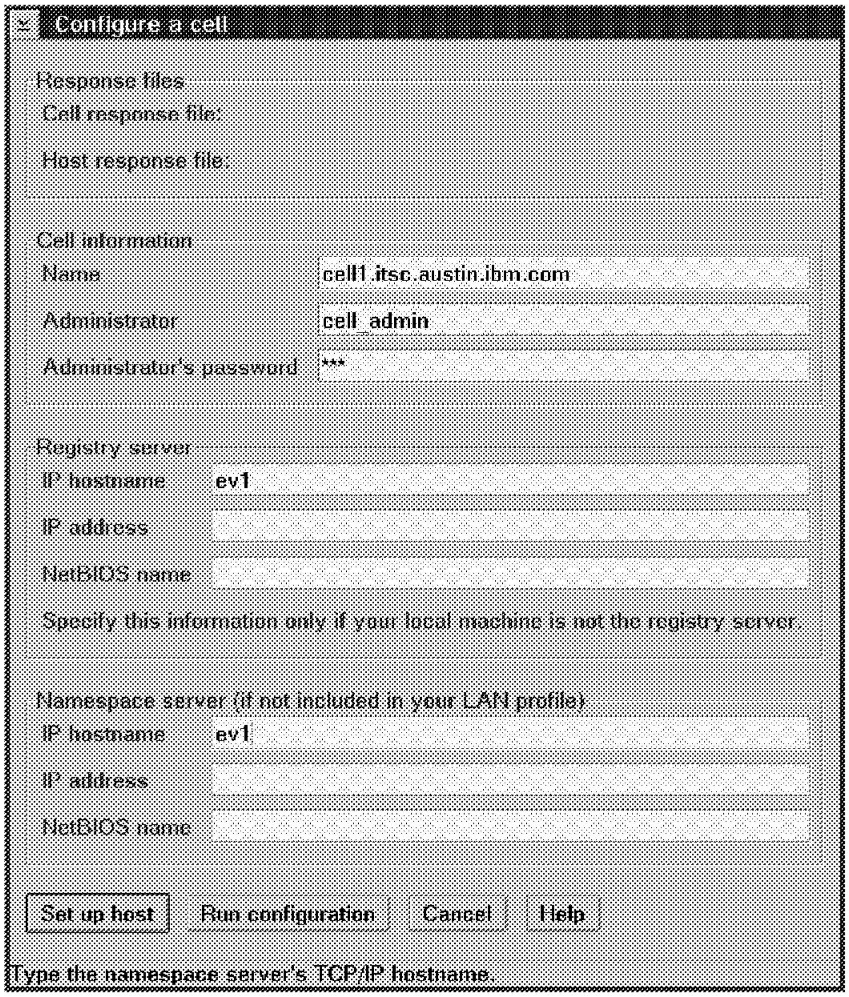
```
[C:\] ucfgdce
```

This is the only way to unconfigure DCE in the current beta version. It stops all running DCE processes and unconfigures the machine. It does not uninstall the DCE code.

To configure DCE on the OS/2 machine, follow this steps:

1. Open the *OS/2 DC v2.1* folder.
2. Double-click on the *Configure DCE* icon (see Figure 45 on page 120).
3. A welcome screen in the *DCE Configuration* window will be displayed.
4. Select **Continue**.

5. The *Specify Configuration Response File Names* window will come up (see Figure 46 on page 120).
6. Click on the **OK** button.
7. The *Specify Configuration Type* window is displayed (see Figure 47 on page 121).
8. Choose **Full**, and click on the **OK** button.
9. The *Configure a Cell* window is displayed shown in Figure 57 below.


Configure a cell

Response files
Cell response file:
Host response file:

Cell information
Name cell1.itsc.austin.ibm.com
Administrator cell_admin
Administrator's password ***

Registry server
IP hostname ev1
IP address
NetBIOS name
Specify this information only if your local machine is not the registry server

Namespace server (if not included in your LAN profile)
IP hostname ev1
IP address
NetBIOS name

Type the namespace server's TCP/IP hostname.

Figure 57. Configure a Cell Window

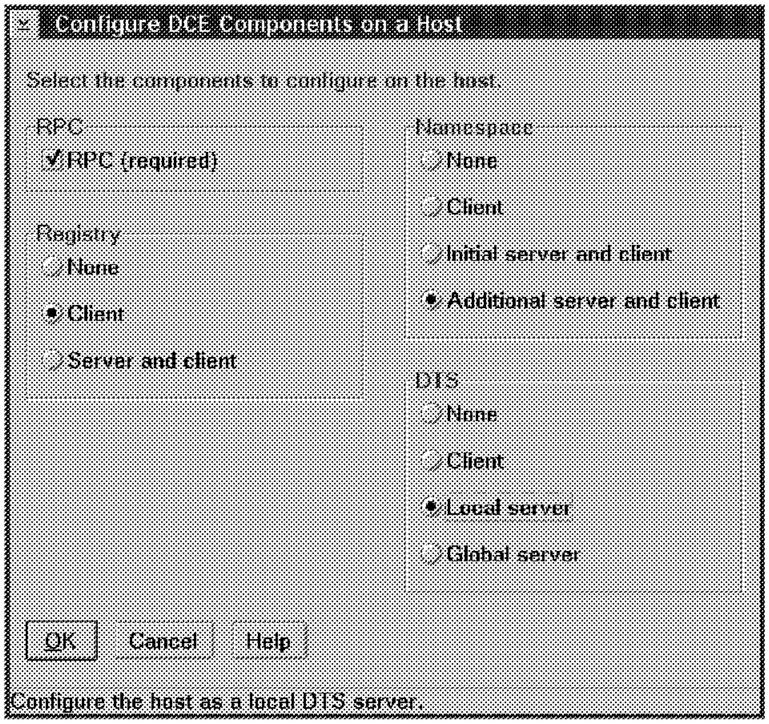
In this screen, we must enter the cell name, the cell administrator′s password, the IP hostname of the security server, and the IP hostname of the namespace server, if it is not in the same LAN. Then click on the **Set up Host** button.

A series of windows described and illustrated in 6.2.3, “Configuring OS/2 with Master Security and Initial CDS Servers” on page 119 come up, which you need to handle as follows:

1. In the *Host Detail* window (see Figure 49 on page 123), you click on the **Edit Settings** button.
2. In the *Set up a Host* window (see Figure 50 on page 124), OS/2 is already selected as the operating system. Fill in the DCE hostname (*sys5*). We also

recommend specifying the name of a server with a good time reference to synchronize the system clock with. Then click on the **Next** button.

3. In the *Select Protocols for a Host* window, the necessary TCP/IP protocols are already selected. Click on the **Next** button.
4. In the *Set up DCE Startup Options for a Host* window (see Figure 52 on page 126), you could choose to automatically start DCE at system startup. Click on the **Next** button.
5. The *Configure DCE Components on a Host* window comes up, as shown in Figure 58 below.


Configure DCE Components on a Host

Select the components to configure on the host.

RPC
✓ RPC (required)

Registry
None [✓ Client] Server and client

Namespace
None Client Initial server and client [✓ Additional server and client]

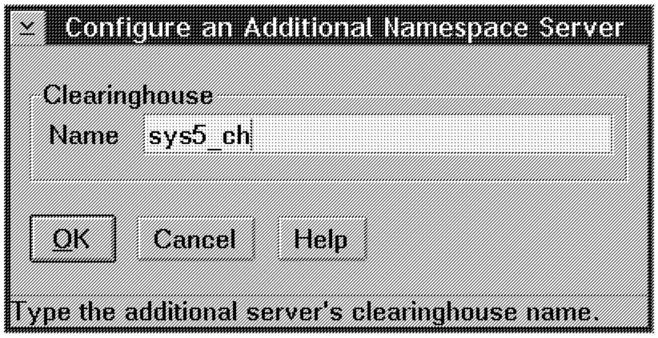
DTS
None Client [✓ Local server] Global server

Configure the host as a local DTS server.

Figure 58. Configure DCE Components on a Host

Select **Client** in the Registry server, **Additional server and client** in the Namespace server and **Local server** in the DTS server group. Then click on the **OK** button.

You will be prompted for the name of the clearinghouse we are creating (see Figure 59 on page 131).



Name sys5_ch

Type the additional server's clearinghouse name.

Figure 59. Configure and Additional Namespace Server

Execute the following steps to complete the configuration:

1. Push the **OK** button.
2. The *Configure a DTS Server* will be displayed (see Figure 55 on page 127).
3. Select the courier role and, if present, fill in the information for an external time provider. Click on the **OK** button.
4. The *Host Details* window is redisplayed (see Figure 49 on page 123).
5. Click on the **OK** button.
6. The *Configure a Cell* window is displayed (see Figure 57 on page 129).
7. Select **Run Configuration**.
8. The *Configuration Progress* window is displayed.

When configuration is complete, push the **OK** button to return to the *DCE Configuration* window and the **Exit** button to end the configuration process.

The DCE is now up and running. From an OS/2 command line window, you can run a dcelogin and other commands to test the environment.

6.3 Setting Up Intercell Communication

To enable intercell communication, we must globally define both cells (either as X.500 or DNS). As shown on Figure 39 on page 103, we have two DCE cells in the same TCP/IP domain (itsc.austin.ibm.com). So, we must define these cells in DNS.

In our environment, we set *ev1* as the DNS server. To enable intercell communication, you will have to define your cells on the DNS server. After registering the cells globally, you must establish a trust relationship between the two cells (see 3.3, “Intercell Authentication” on page 68).

To register *cell1* globally, log in as *root* on *ev1*. Then log in to DCE as the cell administrator. Call smit in the following way:

```
# dce_login cell_admin <password>
# smitty mkdce
   →Register Cell Globally
```

```
                           Register Cell Globally

 Type or select values in entry fields.
 Press Enter AFTER making all desired changes.

                                                      [Entry Fields]
   Name of INPUT File                               []
   Name of named DATA FILE                          [/etc/named.data]



 F1=Help            F2=Refresh          F3=Cancel           F4=List
 F5=Reset           F6=Command          F7=Edit             F8=Image
 F9=Shell           F10=Exit            Enter=Do
```

Fill in the fields with your appropriate file names, and select **Do**. This command will add the following resource records to the named.data file:

```
;BEGIN DCE CELL /.../cell1.itsc.austin.ibm.com INFORMATION
;Initial CDS server
cell1.itsc.austin.ibm.com.              IN      MX      1       ev1.itsc.austin.
ibm.com.
cell1.itsc.austin.ibm.com.              IN      A       9.3.1.68
cell1.itsc.austin.ibm.com.              IN      TXT     ″1 5583b618-f812-11ce-99
ca-10005a4f4629 Master /.../cell1.itsc.austin.ibm.com/ev1_ch 54aa689a-f812-11ce-
99ca-10005a4f4629 ev1.itsc.austin.ibm.com″
;Secondary CDS server
cell1.itsc.austin.ibm.com.              IN      MX      1       ev4.itsc.austin.
ibm.com.
cell1.itsc.austin.ibm.com.              IN      A       9.3.1.123
cell1.itsc.austin.ibm.com.              IN      TXT     ″1 5583b618-f812-11ce-99
ca-10005a4f4629 Read-only /.../cell1.itsc.austin.ibm.com/ev4_ch 657883c2-f813-11
ce-9520-02608c2f0653 ev4.itsc.austin.ibm.com″
;Secondary CDS server
cell1.itsc.austin.ibm.com.              IN      MX      1       sys5.itsc.austin
 .ibm.com.
cell1.itsc.austin.ibm.com.              IN      A       9.3.1.121
cell1.itsc.austin.ibm.com.              IN      TXT     ″1 5583b618-f812-11ce-99
ca-10005a4f4629 Read-only /.../cell1.itsc.austin.ibm.com/sys5_ch 12de6b60-f8f7-1
1ce-98d2-08005aceebf3 sys5.itsc.austin.ibm.com″
;END DCE CELL /.../cell1.itsc.austin.ibm.com INFORMATION
```

The first resource record (of type Mail Exchanger, MX) contains the host name of the system where the CDS server resides. The second resource record (of type Address record, A) contains the address of the system where the CDS server resides. The third record of type TXT, contains information about the replica of the root directory that the server maintains. This information includes the UUID of the cell namespace, the type of replica (*Master*), the global CDS name of the clearinghouse (ev1_ch), the UUID of the clearinghouse, and the DNS name of the host where the clearinghouse resides.

The same information is added for additional CDS servers in the cell. After having added the information to the DNS server′s configuration file, this command refreshes the named daemon. Run the following command to check that the machine *ev3* that runs the global directory agent in *cell2* can resolve the name of *cell1*. On *ev3*, type the following command:

```
# host cell1.itsc.austin.ibm.com
cell1.itsc.austin.ibm.com is 9.3.1.68
```

On Error

If the cell name is not known, make sure the /etc/resolv.conf file on *ev3* points to the name server on *ev1*. Then restart the name daemon on *ev1*:

```
# kill -1 `cat /etc/named.pid`
```

If this does not help, try the following:

```
# stopsrc -s named; startsrc -s named
```

On any machine in *cell2*, log in as root and as cell_admin, and do the following:

```
# cdscp show cell as dns > /tmp/dns_cell2
# cdscp show clearinghouse /.:/* CDS_CHLastAddress >> /tmp/dns_cell2
```

Transfer the dns_cell2 file to the DNS server (*ev1*), and call smit on *ev1* in the following way:

```
# smitty mkdce
   →Register Cell Globally
```

```
                              Register Cell Globally

 Type or select values in entry fields.
 Press Enter AFTER making all desired changes.

                                                          [Entry Fields]
   Name of INPUT File                                    [/tmp/dns_cell2]
   Name of named DATA FILE                               [/etc/named.data]
```

This command will add the information of cell2 to the DNS data file and refresh the DNS server (named process). Note that a temporary file, such as /tmp/dns_cell2, was not necessary in *cell1* because the cell registration on the DNS name server machine has direct access to *cell1* information.

Now, we must establish the ***direct trust peer relationship*** between the two cells. The gdad daemon must be running on both cells. If it is not running, start it with the following command on the system where it resides:

```
# rc.dce gdad
```

On any machine in cell1, run the following command:

```
# dcecp
dcecp> registry connect /.../cell2.itsc.austin.ibm.com -group none \
-org none -mypwd dce -fgroup none -forg none -facct cell_admin -facctpwd dce
```

The registry connect command creates a mutual authentication surrogate in both cells. For more information on intercell authentication, refer to 3.3, "Intercell Authentication" on page 68. You can also create the same trust relationship by running the rgy_edit command as follows:

```
# rgy_edit
Current site is: registry server at /.../cell1.itsc.austin.ibm.com/subsys/dce/se
c/master
rgy_edit=> cell /.../cell2.itsc.austin.ibm.com
Enter group name of the local account for the foreign cell: none
Enter group name of the foreign account for the local cell: none
Enter org name of the local account for the foreign cell: none
Enter org name of the foreign account for the local cell: none
Enter your password:
Enter account id to log into foreign cell with: cell_admin
Enter password for foreign account:
Enter expiration date [yy/mm/dd or 'none']: (none)
```

On Error

If you get an error message, make sure that the GDA is correctly configured on *ev4*.

```
# host cell1
cell1.itsc.austin.ibm.com is 9.3.1.68
```

If this command does not find the address of *ev1*, check the /etc/resolv.conf file on *ev4*. It must point to the name server on *ev1*. Then stop and restart the GDA daemon on *ev4*:

```
# dce.clean gdad
# rc.dce gdad
```

You can now view the contents of the registry database to check the principals that have been created in both cells. Still in *cell1*, you can access the registry service of *cell2* with the following commands:

```
# rgy_edit
Current site is: registry server at /.../cell1.itsc.austin.ibm.com/subsys/dce/se
c/ev2   (read-only)
rgy_edit=> site /.../cell2.itsc.austin.ibm.com
Site changed to: registry server at /.../cell2.itsc.austin.ibm.com/subsys/dce/se
c/master
rgy_edit=> do p
Domain changed to: principal
rgy_edit=> v
nobody                                      -2
root                                         0
daemon                                       1
 ...
dce-ptgt                                    20
dce-rgy                                     21
cell_admin                                 100
krbtgt/cell2.itsc.austin.ibm.com           101
hosts/ev3/self                             102
hosts/ev3/cds-server                       103
hosts/ev3/gda                              104
krbtgt/cell1.itsc.austin.ibm.com           108
```

In the registry of *cell2*, the `registry connect` command has created a principal *krbtgt/cell1.itsc.austin.ibm.com* for *cell1*. You can run the same command on *cell1* to check the principal that has been created for *cell2*. The principal there is *krbtgt/cell2.itsc.austin.ibm.com*.

To further check out intercell access from a machine in *cell1*, you can log into an account of *cell2*:

```
# dce_login /.../cell2.itsc.austin.ibm.com/cell_admin
Enter password:
```

Chapter 7. Migration and Compatibility

This chapter outlines our experience in migrating machines in a cell configured with AIX DCE Version 1.3 running on AIX Version 3.2.5 for the RISC System/6000 to AIX DCE Version 2.1 running on AIX Version 4.1.3 for the RISC System/6000. We will set out the steps that we used to migrate the cell.

If a customer has a one-machine cell and no other AIX machines available, their options for migration are very limited. If, however, a customer has the DCE servers spread over several machines in a cell, they have a couple of options. They can choose to migrate all of the machines in the cell, one after the other, to AIX Version 4.1.3 and then to AIX DCE Version 2.1. The problem here is availability. While a machine is being upgraded, the DCE services it provides are unavailable.

The migration strategy that we employed took advantage of the fact that, by design, machines running OSF DCE 1.0.x and machines running OSF DCE 1.1 may coexist within a cell. We also took advantage of the fact that the primary security server and the initial CDS server can be moved to other machines in the cell. So, we moved the DCE services to another machine in the cell while the original server machine is upgraded. This second approach gives the administrator more flexibility in scheduling the upgrade of the machine, and the server is made unavailable for only a few minutes. The downside of this is that moving servers around can be very complex.

Because servers and clients with different DCE levels can coexist in the same cell, the upgrade process does not have to be performed all at once.

7.1 Compatibility

We tested a mix of DCE servers and clients running at both the AIX DCE 1.3 level (OSF DCE 1.0.3) and the AIX DCE 2.1 level (OSF DCE 1.1) in the same cell, and this is what we found. Servers and clients at both levels can coexist in the same cell. You can even use the dcecp interface from a AIX DCE 2.1 machine to perform a subset of functions on AIX DCE 1.3 machines. The functions you can perform are limited to those you would normally be able to perform with the cdscp or the sec_admin commands. Commands for distributed management of clients and servers are not available for AIX DCE 1.3 machines because the dced daemon is not available on those machines. The value would be in using the dcecp shell (or the Tcl language) to automate or customize some tasks.

Some features of the CDS in OSF DCE 1.1, such a cell name aliases and hierarchical cells, require that you upgrade the Directory Version attribute on the cell root directory to Version 4.0. The default Directory Version for AIX DCE 2.1 is 3.0. At the 3.0 level, the CDS of AIX DCE 1.3 and 2.1 is compatible. AIX DCE 1.3 CDS is not compatible with CDS Directory Version 4.0. For more details, see Chapter 2, “Directory Service” on page 19.

By design, the Security Service provides for incompatibilities between security clients and servers with different DCE levels. For instance, DCE servers on the OSF DCE 1.1 level provide Extended Registry Attributes (ERAs) that are included in the Privilege Attribute Certificates (PACs), now called Extended PACs (EPACs). The EPACs are used to enable authorization checks. Towards OSF DCE 1.0.x clients, the OSF DCE 1.1 security server behaves like an OSF DCE 1.0.x server and does not expect the client to send preauthentication information. In tickets for OSF DCE 1.0.x servers, it does not provide ERAs and EPACs, only PACs. For more details, see Chapter 3, “Security Service” on page 41.

As outlined in 7.2, “Migration” below, we were able to move server functions between machines running AIX DCE at the 2.1 and the 1.3 level. This allows for a greater degree of flexibility when migrating a cell to the new DCE version. Although we used these procedures in a migration process, they have equal value in maintaining the availability of a cell.

7.2 Migration

As outlined above, our strategy was to move the DCE services to other machines to provide continuous DCE availability during the migration. Once the DCE core services are on other machines we upgrade the original DCE servers from AIX 3.2.5 to AIX 4.1.3 and install the new AIX DCE 2.1 code. Then we move the relocated services back to the original system, where AIX DCE 2.1 is able to automatically convert the DCE core service databases to the new format.

Since machines with different DCE levels interoperate, you can also upgrade machine after machine without moving the DCE core services first, if their availability is not required all the time. Begin, for instance, with the security server machines. Once they are upgraded, the cell works with mixed-level machines. Then do the CDS servers if they are on different machines than the security servers. All other DCE nodes can be upgraded at your convenience.

Recommendation

Note that the following procedures in 7.2.2, “Moving the Security Server” on page 137 and 7.3, “Moving the Initial CDS Server” on page 138 require actions on *all* machines in the cell and are relatively complex. The CDS server swap is only easy if there is only one CDS server defined, which is very unlikely. If the CDS configuration is more complex, the procedure needs modification.

If you want the target servers to take over the new roles for good, it might be worth while to do the swap. If you wanted the old servers to eventually become the new servers again, you would need to perform these steps twice. So, it might be much more convenient to make system backups as a fallback option, and reserve a weekend or night to upgrade a set of machines in one shot without having to move servers around. In case of problems, you restore the old configuration. The upgrade is explained in 7.2.1, “Using the AIX 4.1 Migration Utilities” below.

7.2.1 Using the AIX 4.1 Migration Utilities

We upgraded an AIX Version 3.2.5 systems to AIX 4.1 using the migration option. After the base operating system was at the new level, we then had to install the DCE for AIX 2.1 code. When the installation completed, we found that DCE had been upgraded to the 2.1 level, and all of our data remained intact. We also found the installation procedure did not remove the reference to AIX 1.3 code from the object data manager. This was easily remedied by using `smit install_remove` to remove the reference.

7.2.2 Moving the Security Server

The following procedure was taken from the DCE 1.3 documentation. We tested this procedure to move the security server from an DCE 1.3 machine to a machine in the same cell running DCE 2.1. The procedure is provided here for your convenience:

1. Add a Version 4.1.3 machine with DCE 2.1 to the existing cell as a client.
2. Configure the Version 4 machine as a secondary security server.
3. On the original primary security server, do the following:
   a. Update the rpccp cache on /.:/sec:
   ```
   # dce_login cell_admin <password>
   # rpccp show group /.:/sec -u
   # sec_admin -s /.:/subsys/dce/sec/master
   sec_admin> state -m
   sec_admin> state -s
   sec_admin> quit
   ```
   b. Back up the registry database and support files:
   ```
   # cd /opt/dcelocal/var/security
   # tar <tarfile> ./.mkey ./rgy_data
   ```
   The file <tarfile> is any file or device that can be used to transfer data to the new primary server.

   c. Stop the secd daemon:
   ```
   sec_admin> stop
   ```
   d. Remove the registry database and support files on the original primary machine:
   ```
   # rm /opt/dcelocal/var/security/.mkey
   # rm -r /opt/dcelocal/var/security/rgy_data
   # rm -r /opt/dcelocal/var/security/rgy_data.bak
   # rm /opt/dcelocal/var/security/tmp/krb5kdc_rcache
   ```
   e. Comment-out the line in /etc/rc.dce that will restart the secd when /etc/rc.dce is executed:
   ```
   from:    daemonrunning $DCELOCAL /bin/secd
   to:      # daemonrunning $DCELOCAL /bin/secd
   ```
   f. Delete the line in /etc/mkdce.data that indicates the primary security server is running on the local machine:
   ```
   sec_srv       COMPLETE   Security Server
   ```
4. On the NEW primary server, perform the following steps:
   a. Remove references to the old server in the namespace:
   ```
   # dce_login cell_admin <password>
   # cdscp delete obj /.:/subsys/dce/sec/master
   # rpccp remove member /.:/sec -m /.:/subsys/dce/sec/master
   # rpccp remove member /.:/sec-v1 -m /.:/subsys/dce/sec/master
   ```
   b. Destroy the secondary machine:
   ```
   # sec_admin -s /.:/subsys/dce/sec/repl1
   sec_admin> destroy subsys/dce/sec/repl1
   sec_admin> quit
   ```

c. Verify that the /opt/dcelocal/var/security/rgy_data directory is empty, and restore the backed-up files from the original primary machine to the /opt/dcelocal/var/security directory:

```
# cd /opt/dcelocal/var/security
# tar -xvf <tarfile>
```

Verify that the permission set on the restored rgy_data directory is 755, and the permission on all restored files (.mkey and all files in the rgy_data directory) is 600.

5. On every machine in the cell, do the following:

   a. Edit the /opt/dcelocal/etc/security/pe_site file, and change the IP address to that of the new primary server.

   b. Edit the /krb/krb.conf file, and update the entry to reflect the new security server machine (change machine name).

6. On the new primary machine, do the following:

   a. Start the secd by issuing the command :

   ```
   # secd -d -v
   ```

   If secd does not restart, try using ″dce.clean secd″; wait three minutes, and try again. Our experience was that it did not work until we did the dce.clean.

   b. Log in as cell_admin, and perform the following steps to complete the clean-up of the old secondary machine:

   ```
   # dce_login cell_admin <password>
   # sec_admin -s /.:/subsys/dce/sec/master
   sec_admin> delrep subsys/dce/sec/repl1 -f
   sec_admin>quit
   ```

7. On every machine in the cell, except for the new primary, kill all of the CDS daemons currently active on the system. If you don′t perform this step, the sec_clientd might not restart!

   ```
   # cd /opt/dcelocal/var/adm/directory/cds
   # rm cds_cache.*
   # rm cds_clerk_*
   # rc.dce cds
   # rpccp show group /.:/sec -u
   # rpccp show group /.:/sec-v1 -u
   ```

8. On every machine in the cell, stop and restart sec_clientd so that it will re-bind to the new primary machine. Use the -purge option so that existing credentials will be destroyed and re-created.

   ```
   # dce_clean sec_clientd
   # sec_clientd -purge
   ```

7.3 Moving the Initial CDS Server

This procedure has been used to move the CDS server in an AIX DCE 1.3 environment and was successfully tested in moving a CDS server from an AIX DCE 1.3 machine to an AIX DCE 2.1 machine.

Note: This procedure only works straight forwardly if no second CDS server has already been defined. If one or more CDS servers are active and master replicas are spread over different clearinghouses, this procedure needs modifications. When defining the replica set, you need to specify all existing replicas.

This is the procedure we tested:

1. Create an additional CDS server on the machine you want to become the initial CDS server.

   ```
   # mkdce cds_second or
   # smit mkcdssrv
   ```

2. Log in to DCE as cell_admin on both machines.

3. Use the cdsli command to verify the directories in CDS. We will copy all directories from BoxA to BoxB.

   ```
   # cdsli -dR
   ```

4. Verify that the additional CDS server is empty on BoxB.

   ```
   # cdscp
   cdscp> set cdscp preferred clearinghouse /.:/<BoxB>_ch
   cdscp> show dir /.:/*
   cdscp> quit
   ```

5. Replicate all CDS directories to the new additional CDS server′s (BoxB) clearinghouse, and make them the master replicas:

   ```
   # for dir in $(cdsli -R); do
   > echo ″Creating replica for $dir″
   > cdscp create replica $dir clear /.:/<BoxB>_ch
   > echo ″Swapping master CDS attribute for $dir″
   > cdscp set dir $dir to new epoch master /.:/<BoxB>_ch readonly /.:/<BoxA>_ch
   > done
   ```

6. Verify that the swap worked. You should see that the master replica for everything, including the /.: directory located on BoxB′s clearinghouse:

   ```
   # cdscp
   cdscp> show dir /.:
   cdscp> show dir /.:/*
   ```

7. At this point, everything has been moved to the new CDS server (BoxB). All entries in the CDS server on BoxB are the master replica entries. You have swapped the initial CDS server to the additional CDS server and vice versa. Now we must modify the /etc/mkdce.data file to reflect the change. Edit the /etc/mkdce.data file on BoxA to change the line:

   ```
   from:   cds_srv     COMPLETE   Initial CDS Server
   to:     cds_second  COMPLETE   Additional CDS Server
   ```

   Edit the /etc/mkdce.data file on BoxB to change the line:

   ```
   from:   cds_second  COMPLETE   Additional CDS Server to
   to:     cds_srv     COMPLETE   Initial CDS Server
   ```

8. Delete the CDS cache on both machines:

   ```
   # dce.clean cds
   # cd /var/dce/adm/directory/cds
   # rm cds_cache.*
   # rc.dce cds
   ```

9. It is a good idea to stop and restart DCE on the two machines you have just worked with:

   ```
   # dce.clean all
   # rc.dce all
   ```

10. Remove the additional CDS server from BoxA. (Be sure that you are logged into DCE as cell_admin):

    ```
    # rmdce cds_second
    ```

11. Then perform a CDS client cache refresh on *all* systems in the cell:

    ```
    # dce.clean cds
    # cd /var/dce/adm/directory/cds
    # rm cds_cache.*
    # rc.dce cds
    ```

Chapter 8. High-Performance and High-Availability Configurations

Depending on the number of users, the amount of DCE applications and application servers, the number of DCE clients, and the size of a cell in general, DCE can impose heavy load on the Security and CDS server machines. Users, DCE clients and DCE servers all need several kinds of tickets from the Security Service to authenticate themselves and to run authenticated RPCs. DCE clients need information from CDS when they look for application servers, and application servers export their interfaces when they start and should unexport them when they terminate.

So, we need fast machines for these DCE Core Services, and we also need them to be highly available. Although DCE Core Services are all replicated, which improves performance and availability, there are circumstances where replication is not sufficient:

- Applications exporting and unexporting their interfaces need access to a read/write replica of CDS.
- In a network topology with many remote sites and relatively slow communication links, replication might introduce additional bottlenecks if requests go to the ″wrong″ replication servers. Careful planning is necessary to avoid this situation.

The RISC System/6000 is the most advanced platform on which to run DCE Services. It provides the following options to improve performance and availability in situations where replication is not sufficient:

- ***IBM AIX High Availability Cluster Multi-Processing*** — DCE core servers and Distributed File System (DFS) servers can be put on an HACMP cluster. These servers run on one node, and when this node dies, the other node can take over. The IP address, and even the LAN hardware address of the server, remains unchanged. So, the base DCE functions are permanently read/write accessible.
- ***IBM RS/6000 SMPs*** — Since DCE servers are inherently multithreaded, they can all take advantage of shared-memory symmetric multiprocessor machines with the new DCE Version 2.1 on AIX Version 4.
- ***IBM SP2*** — For DCE, this machine can be considered as multiple distinct machines with a very fast communication link. So, installing DCE servers which have to communicate a lot with each other on different SP2 nodes might be helpful. This can be core servers, DFS servers and application servers. HACMP can also be run between nodes in an SP2.

The following two sections give an overview of the basic concepts and terms used in discussing multiprocessor machines and HACMP.

Please Note

We did not have the opportunity to test these configurations. However, we would like to share with you our assumptions based on theory so that you understand the concept. Please watch for official announcements as to whether or not these are supported and tested configurations.

8.1 IBM AIX High Availability Cluster Multi-Processing

IBM AIX High Availability Cluster Multi-Processing represents an important product that combines software and hardware to minimize down time by quickly restoring services when a system, a component or an application fails. While not instantaneous like fault-tolerant or continuous-availability systems, restoring services is rapid and usually takes only a couple of minutes.

HACMP features are used in business-critical applications, such as order processing, debit/credit transactions in banking or hotel reservations and others. Particularly in conjunction with RAID disks, HACMP provides a stable environment for Relational Database Management System (RDBMS) applications.

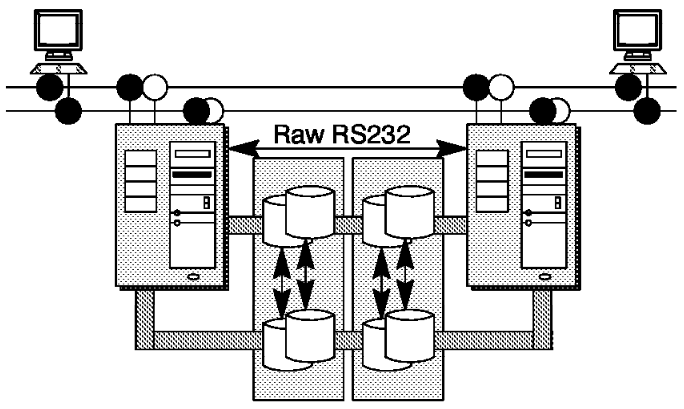


Figure 60. The HACMP/6000 Environment

HACMP can be configured to provide no single point of failure. To achieve this and operate correctly, HACMP requires extra hardware, such as two TCP/IP link attachments per cluster node, a direct serial connection, and shared (twin-tailed) SCSI or serial disks.

8.1.1 Resources and Resource Groups

The basic concept of HACMP Version 4.1 is that of resources and resource groups as entities that can be taken over by another node in the cluster. A ***resource*** can be one of the following:

- Disks
- Volume Groups
- File systems (including NFS-mounted or exported file systems)
- Service IP addresses
- Applications

Each resource in a cluster is defined as part of a ***resource group***. This allows you to combine related resources that need to be together to provide a particular service. A resource group also includes the list of nodes that can acquire those resources and serve them to clients. A resource group is defined as one of three types:

- ***Cascading Resource Groups*** — All nodes in a cascading resource group are assigned priorities for that resource group. The resources always cascade to the highest priority node joining or reintegrating into the cluster.
- ***Rotating Resource Groups*** — A rotating resource group is associated with a group of nodes, rather than with a particular node. As participating nodes join the cluster, they acquire the first available rotating resource group until all groups are acquired. A node can only possess a maximum of one rotating resource group per network. The remaining nodes remain in a standby mode. When a node holding a resource group leaves the cluster, the node with the highest priority and available connectivity takes over. A reintegrating node remains in standby if all resource groups are currently owned; it does not take back the resources that it had initially served.
- ***Concurrent Resource Groups*** — May be simultaneously shared by multiple nodes. Resources that can be part of a concurrent resource group are limited to volume groups with raw logical volumes, and raw disks. The disks involved must be RAID 9333-xx1 or SSA disks. When a node fails, there is no takeover involved. On reintegration, a node once again accesses the resources simultaneously with the other nodes.

8.1.2 HACMP Cluster Configurations

Combining its ability to integrate up to eight nodes in a cluster with the concept of resource groups makes HACMP very flexible. Some of the basic failover configurations of HACMP are as follows:

- ***Non-Concurrent Disk Access Configurations*** — Disk resources are owned and accessed by one machine which has the highest priority for that resource at any one time. Four different variations are:
  - ***Hot standby*** — Assuming that there is a single (cascading) resource group in the cluster, a node 1 has priority for it over node 2. Node 2 may be idle or may be providing non-critical services which it can give up. It is therefore referred to as a hot standby node. When node 1 fails or has to leave the cluster for a scheduled outage, node 2 acquires the resource group and starts providing the service. When node 1 reintegrates into the cluster, it takes back the resources. The advantage of this type of configuration is that you can shift from a single-system environment to an HACMP cluster at a low cost by adding a less powerful processor to serve in a failover situation.
  - ***Rotating standby*** — A similar hardware configuration as the hot standby configuration. However, since the resources are defined as rotating, node 2 keeps ownership of the shared resources after a failover when node 1 rejoins the cluster. Node 1 becomes the standby node in this case. Unlike in the hot standby mode, node 1 and node 2 should be machines of equal power. They provide better availability and performance than a hot standby configuration.
  - ***Mutual takeover*** — For this configuration, we assume that we have two cascading resource groups and two nodes. Each node has priority over the other for one of the resource groups. So, both machines are doing work and are watching each other, ready to take over the other′s resources (disks and service IP address). In case of a failover, one machine performs the work previously spread over two machines.
  - ***Third-party takeover*** — The machines that are performing critical services just take care of the resource group(s) for which they have

priority 1 and have no resource group(s) assigned with a lower priority. This means they are not configured for taking over a failing machine′s resources. A third machine acts as a standby (priority 2 or lower) for the resource groups of two (or more) productive machines, ready to take over the resources if one (or more) of them fails.

- ***Concurrent Disk Access Configurations*** — They actually share the disks and are able to concurrently access them. Special daemons control and serialize competing disk access requests. Programs running in this mode, such as the ORACLE Parallel Database Server, have to use the special locking API provided by those daemons.

8.1.3 Benefits for DCE

AIX DCE and Encina services can basically run on non-concurrent configurations, with some restrictions. Since HACMP has become much more flexible, we must come up with some rules:

1. Everything which makes up a certain DCE node must be configured into the same resource group. This includes the /var/dce and /etc/dce file systems as well as the service IP address of the DCE node. It may also include application-specific file systems.
2. Configure the cluster so that multiple resource groups or nodes containing DCE services cannot be taken over by a single node.

What this means is that you must make sure that two different DCE machines are not accidentally migrated to one single machine in a failover situation. If you have a mutual takeover configuration, as described above, only one machine should be configured as a DCE node. If you have third-party takeover, only one of the productive machines that can be taken over by a specific, third machine should be configured as a DCE node.

Both security and CDS servers can be replicated within DCE, but if the system containing the master database fails, write access is not possible anymore. Tickets can be issued from a slave security server, thus leaving the cell operational. However, if for any reason write access to the security registry is always required, the master security server must be in an HACMP cluster. CDS is a little bit different. When application servers start, they export their binding information into the CDS namespace. So, running CDS in an HACMP cluster always makes sense, if the directories to which write access is needed are not distributed.

8.2 Multiprocessing

In the past, improvements in computer performance have been achieved simply by designing better, faster uniprocessor (UP) machines. However, UP designs have built-in bottlenecks. The address and data bus restrict data transfers to a one-at-a-time trickle of traffic. The program counter forces instructions to be executed in strict sequence. It now appears that further significant performance gains will need a different design. A way to increase performance is through a multiprocessor (MP) architecture. However, MP architectures do not just help to improve performance. Scaling-up a system (by putting more processors in it), rather than replacing it, is also a good way to protect the investment.

8.2.1 Multiprocessor Architectures

Multiprocessing can be categorized in a number of ways, but some of the more important aspects to consider are:

1. Do the processors share resources, or do they each have their own? Resources to consider include the operating system, memory, input/output channels, control units, files, and devices.
2. How are the processors connected? They might be in a single machine sharing a single bus, connected by other topologies (like switch, ring), or they might be in several machines using message-passing across a network.
3. Will all the processors be equal, or will some of them be specialized? For instance, all the processors might be able to do integer arithmetic, but only one of them can do floating point.
4. Will parallel programming be supported? The act of sharing parts of a program represents an extra task in itself.
5. Will it be easy to enhance/upgrade the system at a later date? Usually, the addition of a new processor will not cause system throughput to increase by the rated capacity of the new processor because there is additional operating-system overhead and increased contention for system resources.
6. What happens if one of the processors fails? One of the most important capabilities of MP operating systems is their ability to withstand equipment failures in individual processors and to continue operation.

Sometimes MP environments are classified with the terms *loosely coupled multiprocessing* and *tightly coupled multiprocessing*. However, this usually only relates to the type and speed of the network connecting the nodes together that perform some common work. The following classification is more useful:

- ***Shared-Nothing MP*** — Each processor is a stand-alone machine, and each one has its own caches, memory and disks. Also, each processor runs a copy of the operating system. Processors can be interconnected via a LAN if they are loosely coupled or via a switch if they are tightly coupled. Communication between processors is done via a message-passing library. Examples of shared-nothing MPs are the ***IBM SP1 and SP2*** as well as Tandem, Teradata and most of the so-called *massively parallel* systems.
- ***Shared-Disks MP*** — Each processor has its own caches and memory, but disks are shared. Also, each processor runs a copy of the operating system. Processors can be interconnected through a LAN or a switch. Communication between processors is done via message passing. Examples of shared-disks MPs are IBM RS/6000 with HACMP and DEC VAX-cluster.
- ***Shared-Memory Cluster*** — Each processor has its own caches, memory and disks and runs a copy of the operating system. Processors are interconnected via a piece of shared memory over which communication is done. This is not a very widespread form of MP.
- ***Shared-Memory MP*** — All of the processors are tightly coupled inside the same box with a high-speed bus or switch between the processors, the I/O subsystem and the memory. Each processor has its own caches, but shares the same global memory, disks and I/O devices. Only one copy of the operating system runs on all processors. It means that the operating system has to be designed to exploit this type of architecture. There are two basic operating-system organizations for shared-memory MPs:

1. ***Master/slave organization (asymmetric)*** — One processor is designed as the master and the others are the slaves. The master is a general-purpose processor and performs input and output operations as well as computation. The slave processors perform only computation. Utilization of a slave may be poor if the master does not service slave requests efficiently enough. I/O-bound jobs may not run efficiently since only the master executes the I/O operations. Failure of the master is catastrophic.

2. ***Symmetric multiprocessing organization*** — All of the processors are functionally equivalent and can perform I/O and computational operations. The operating system manages a pool of identical processors, any of which may be used to control any I/O devices or reference any storage unit. A process may be run at different times by any of the processors, and at any given time, several processors may execute operating-system functions in kernel mode.

Shared memory MP with symmetric organization (or ***SMP***) is one of the most common multiprocessing implementations on the commercial market. Examples of this implementation are the ***IBM G30, J30 and R30*** models.

8.2.2 MP-Safe Programming

AIX Version 4 can make use of SMP machines because it supports kernel-level threads and can dispatch threads to distinct processors. When programming multithreaded applications, you must consider serialization of access to resources. A program is ***thread-safe*** when multiple threads in a process can be running that program successfully without data corruption. A library is reentrant or thread-safe when multiple threads can be running a routine in that library without data corruption. To be thread-safe, programs use reentrant libraries and implement locking schemes to prevent data from being accessed by multiple threads at the same time. See also Chapter 11, “Threads” on page 209 for more details on threads.

To be ***MP-safe***, a multiprocess program or a multithreaded program must serialize access to shared resources by using a global lock mechanism. However, the lock mechanism used in a UP environment may not be sufficient in an MP environment. In particular, programs designed for UPs that spawn off multiple processes must be ported to MP environments, unless they make themselves run only on a specific processor of an MP. They need to use a special locking API for MPs. Thread-safe programs are usually MP-safe.

A program is ***MP-efficient*** when it is MP-safe and when it spends a minimum time dealing with locks. Finer granularity locks usually improve the efficiency because they decrease the waiting time for other threads. However, overly fine-granular locks may lead to a higher number of calls to the locking API, which takes up a considerable amount of CPU cycles. So, a good balance must be found for the degree of granularity.

8.2.3 Benefits for DCE

The use of multiple threads concurrently running in an SMP system improves the performance of an application. This is true for RPC servers as well as for RPC clients. Using multiple threads allows an RPC client or server to make or serve multiple concurrent RPC calls. Application threads can continue processing independently of RPCs.

Chapter 9. DCE Control Program and Tcl

The DCE Control Program (the dcecp command) is the administrator tool in OSF DCE 1.1 that integrates the functions of several tools used in OSF DCE 1.0.x, such as cdscp, rpccp, rgy_edit, acl_edit, and dtscp.

The dcecp language is based on the Tool Command Language, generally known as ***Tcl*** (pronounced ″tickle″). The dcecp commands are implemented as Tcl commands in a Tcl interpreter.

This chapter introduces Tcl and explains how to create dcecp programs and extensions.

9.1 What is Tcl?

Tcl (Tool Command Language) is a freely distributable, simple interpreted language designed to be used as a common extension and customization language for applications. It was designed and implemented by Dr. John Ousterhout in the hope that application designers could spend more of their time on applications and less on scripting languages and in the hope that users could spend less time learning new scripting languages for each new application. Many useful applications, some of them sold commercially, use Tcl as their scripting language.

Tcl is clean, regular and relatively easy to learn. It is command-oriented, and commands added by applications and users exist on an equal footing with the built-in Tcl commands. Tcl has both simple variables and associative arrays (tables), and all values (including procedure bodies) are represented as strings. Simple customization scripts (such as preference initialization scripts) usually look much like novice users expect them to: a series of simple commands which set options.

Tcl is implemented as a C library that can be embedded in an application. The application can add its own commands to the interpreter (using a clean C interface). It is distributed under a license which allows use for any purpose with no royalties.

The Tcl library consists of a parser for the Tcl language, routines to implement the Tcl built-in commands and procedures that allow each application to extend Tcl with additional commands specific to that application. The application program generates Tcl commands and passes them to the Tcl parser for execution. Commands may be generated by reading characters from an input source or by associating command strings with elements of the application′s user interface, such as menu entries, buttons or keystrokes. When the Tcl library receives commands, it parses them into component fields and executes built-in commands directly. For commands implemented by the application, Tcl calls back to the application to execute the commands. In many cases, commands will invoke recursive invocations of the Tcl interpreter by passing in additional strings to execute (procedures, looping commands and conditional commands all work in this way).

An application program gains three advantages by using Tcl for its command language:

- Tcl provides a standard syntax — Once users know Tcl, they will be able to issue commands easily to any Tcl-based application.
- Tcl provides programmability — All a Tcl application needs to do is to implement a few application-specific, low-level commands. Tcl provides many utility commands plus a general programming interface for building up complex command procedures. By using Tcl, applications need not re-implement these features.
- Extensions to Tcl, such as the Tk toolkit, provide mechanisms for communicating between applications by sending Tcl commands back and forth. The common Tcl language framework makes it easier for applications to communicate with one another.

Note that Tcl was designed with the philosophy that one should actually use two or more languages when designing large software systems. One for manipulating complex internal data structures, or where performance is key, and another, such as Tcl, for writing small scripts that tie together the C pieces and provide hooks for others to extend. For the Tcl scripts, ease of learning, ease of programming and ease of gluing are more important than performance or facilities for complex data structures and algorithms. Tcl was designed to make it easy to drop into a lower language when you come across tasks that make more sense at a lower level. In this way, the basic core functionality can remain small, and one need only bring along pieces that one particularly wants or needs.

9.2 dcecp Introduction

Many people seem to get confused trying to understand the relationship between dcecp and Tcl. It is a close relationship, but understanding which commands are Tcl commands and which are DCE commands is unimportant. The dcecp language is implemented as a Tcl extension and supports all Tcl commands.

A user of dcecp starts the control program and enters interactive commands at the dcecp> prompt. There are other use models, such as writing scripts and entering commands from the shell command line, but they are less common. An important usability feature of dcecp is that it provides command-line recall and editing similar to ksh's Emacs mode. At the prompt, the user can enter base Tcl commands, such as:

```
dcecp> echo foo
foo
dcecp> lsort {a c e b d}
a b c d e
dcecp> foreach i [lsort {a c e b d}] {puts stdout $i}
a
b
c
d
e
```

The Tcl library implements an interpreter that understands all the base Tcl commands. Any program using this library has access to these same commands. A typical Tcl program usually implements new project-specific commands. Not surprisingly, dcecp implements commands that manipulate DCE data. Users can, at the same dcecp> prompt, enter DCE commands, such as:

```
dcecp> principal create nordpol -fullname {Norbert Nordpol}
dcecp> acl modify /.:/hosts -add {user nordpol -rwx}
dcecp> object delete /.:/hosts/gandalf
```

Note: The only part of DCE that uses Tcl is dcecp. The Tcl library is not part of libdce. The goal of DCE is to ship dcecp, not Tcl.

To the user of dcecp, there is no visible difference between base a Tcl command and a command that manipulates DCE data. They might see the base Tcl commands when running other programs that use Tcl, but they will only see the DCE commands in dcecp. The Tcl and DCE commands can be combined (output not shown for space reasons):

```
dcecp> foreach i [dir list /.: -directories] {dir show $i}
```

This allows an advanced user to write extensible scripts, a feature that is very important.

9.3 Tcl Language Components

This section explains Tcl basics which are needed to develop extensions to the dcecp command. It can be followed as a tutorial.

9.3.1 Tcl Language Syntax

The syntax of Tcl commands is simple:

```
command_name arg1 arg2 ....
```

command_name is the name of a built-in command or a procedure. To terminate a command, you use a newline character or a semicolon. The words in the command are separated by spaces or tabs.

Tcl executes a command in two steps: parsing and execution. In the parsing step, the Tcl interpreter divides the command into words and performs variable and command substitutions. In the execution part, Tcl assigns meanings to the words. Tcl searches for a built-in command or procedure that matches the first word. If a command is found, it is invoked with all the words of the command passed as arguments.

Tcl provides two forms of substitution: variable and command substitution. Variable substitution is performed when the parser encounters a dollar sign. The value of the variable is then inserted into the command word (See 9.3.2, “Variables” on page 150). Command substitution causes part or all of a command word to be replaced with the result of another Tcl command. Command substitution is invoked by enclosing a nested command in brackets: [command]. This causes command to be executed and its result to be inserted in the command word. To print the special characters, such as the dollar sign and bracket, you can use the backslash. The backslash can also be used to insert special characters, such as newlines or backspaces. Examples of substitution are:

```
dcecp> set server hosts/ev7/self
hosts/ev7/self
dcecp> set server_attr [principal show $server]
{fullname {}}
{uid 102}
{uuid 00000066-edbe-21ce-9000-10005aa86e2d}
{alias no}
```

```
{quota 0}
{groups none subsys/dce/dts-servers}
dcecp> set total 15.50
15.50
dcecp> puts stdout "Total is \$$total"
Total is $15.50
```

The first command assigns the value hosts/ev7/self to the server variable. The second command assigns the result of executing the principal command over the principal stored in the server variable to the server_attr variable. The last two commands show the use of the backslash to print a dollar sign.

9.3.2 Variables

Tcl supports two kinds of variables: simple variables and associative arrays. The set command is used to define variables of any type. This command takes two arguments, the name of the variable and its value. If only the variable's name is passed, it returns its value. Once you have established a value for a variable, it can be used elsewhere in your script. The dcecp program uses the dollar sign to trigger insertion of the current value into the command word. Some simple examples are:

```
dcecp> set a 7
7
dcecp> expr $a+2
9
dcecp> set CDS_DirectoryVersion 4.0
4.0
```

To remove a variable, use the unset command:

```
dcecp> unset a
dcecp> set a
Error: can't read "a": no such variable
```

Arrays are explained in 9.3.3.2, "Arrays" on page 152.

9.3.2.1 Expressions

Expressions are useful for things, such as comparing numeric information, setting thresholds for monitoring purposes or producing statistical information. Expressions combine values with operators to produce new values. The expr command takes one argument: the expression. In comparison operations, Tcl returns 0 for false and 1 for true. For example:

```
dcecp> set x 12
12
dcecp> expr $x + 4
16
dcecp> expr {$x <= 3}
0
dcecp> expr sin($x) * 3.1415
-1.68564
```

The dcecp program normally treats numbers as integers or real numbers. Integers are usually specified in decimal, but if the first character is 0, then the number is read in octal. For hexadecimal interpretation precede the number with 0x. Numbers can also be represented in the format specified by the ANSI C standard, such as 3.45e-3. Tcl supports operators similar to the operator expressions in ANSI C, which are: *, /, %, +, -, <<, >>, <, >, <=, ==, !=, &,

¬, |, &&, and ||. It also includes math functions, such as sin(), cos() and tan().

Strings can also be compared with the expr command. You must quote the string value so the expression parser can identify it as a string. For example:

```
dcecp> set x foo
foo
dcecp> if { $x == "foo" } {puts foo}
foo
dcecp> expr {$x < "goo"}
1
```

9.3.3 Data Structures

In Tcl, there is only one data type, which is strings. All commands, expressions, variable values, and procedure return values are strings. In addition, there are two higher-level data structures: lists and arrays. Lists are implemented as strings. Their structure is defined by the syntax of the string. Some Tcl commands interpret the strings in different manners. For example, expr treats its *argument* as an expression. The argument it receives is a string. If the argument is 4 + 5, the returned result is the string 9.

9.3.3.1 Strings

A string is a sequence of characters. The chief command to manipulate strings is string. The string command is actually about a dozen string manipulation commands rolled into one. Its syntax is:

```
    string <operation> <string_value> <args>
```

Some examples of string command operations are:

compare <str1> <str2>	Compares strings. Returns -1 if <str1> is lexicographically less than <str2>, 0 if equal and 1 if greater.
first <str1> <str2>	If <str1> is contained in <str2>, it returns the index of the first occurrence or -1 otherwise.
last <str1> <str2>	If <str1> is contained in <str2>, it returns the index of the last occurrence or -1 otherwise.
length <string>	Returns the number of characters in <string>.
match <pattern> <str>	Returns 1 if <pattern> matches <str> using global style matching rules. This pattern matching is similar to that of UNIX shells. For example, a*.[ch], matches all words that begin with "a" and end with ".c" or ".h".
range <string> <i> <j>	Returns a substring from <string> that lies between the indices given by <i> and <j>. The third argument can be the keyword end to indicate the last character of the string.

Another command used when working with strings is the format command, which is similar to the sprintf() function in C. Its syntax is:

```
    format <spec> <value1> <value2> ...
```

The <spec> argument is a format string which may contain a list of conversion specifiers enclosed in quotes, such as "%.3f". For each specifier in the format list, a replacement string is inserted which consists of the next value of the argument list reformatted according to the conversion specifier. The result is the specification string, with each conversion specifier replaced by the corresponding replacement string. Format supports almost all of the conversion specifiers defined for the sprintf() function in C. For example:

```
dcecp> clock show
1995-07-03-10:49:10.028-05:00I-----
dcecp> set time [string range [clock show] 11 18]
10:49:27
dcecp> format "Time of the day is: %s" $time
Time of the day is: 10:49:27
```

9.3.3.2 Arrays

Tcl implements one-dimensional arrays. However, since an array's index is a string, you can "simulate" multi-dimensional arrays by concatenating multiple indices into a single element name. The index of an array is delimited by parentheses. The following example simulates a two-dimensional array:

```
dcecp> set a(1,1) 1
1
dcecp> set a(1,2) 3
3
dcecp> set i 1 ; set j 2
2
dcecp> set a($i,$j)
3
```

The array command can be used to obtain information about an array. The following commands inquire the number of elements (size) and the forthcoming indices (names) of an array:

```
dcecp> array size a
2
dcecp> array names a
1,1 1,2
```

The array command can also be used to iterate through array elements. The startsearch option returns a search identifier key. Using this search identifier key, the nextelement option returns a string representing the first (next) index name. The donesearch option terminates the search and frees any resources used by the search. This is shown in the following example:

```
dcecp> array startsearch a
s-1-a
dcecp> array nextelement a s-1-a
1,1
dcecp> set a([array nextelement a s-1-a])
3
dcecp> array donesearch a s-1-a
```

9.3.3.3 Lists

Some Tcl commands interpret their string arguments as a list. In Tcl, a list consists of elements separated by white spaces (space, tab or newline). The following three examples are lists which have three elements:

```
    tom dick harry
    a b c
    1 2 !
```

There are three metacharacters that are treated specially when parsing lists. They are:

{ } Braces are used for grouping. If a left brace is encountered, then the element is terminated by the matching right brace, instead of the next whitespace. Braces nest. The following are also examples of lists with three elements:

```
    tom {dick harry} {sue mary jane}
    a {b {c d e}} f
```

However, some elements in the above example are lists or even nested lists.

″ ″ Double quotes are also used for grouping. However, there are two differences between them and braces. First, they do not nest. Second, Tcl will perform command, variable and backslash substitutions for items within double quotes. Tcl does not perform these substitutions when braces are used.

\ Backslashes are used to escape metacharacters and to insert non-printing characters. Standard substitutions occur. `\t` is a tab character, and `\n` is a newline. Braces and double quotes can be inserted in lists with backslashes as follows:

```
    a b\ c d
    a b \}
```

Again, these example lists all have three elements.

The net effect of using these metacharacters and Tcl commands on a command line is that lists appear as elements surrounded by braces. The fact that the top level of braces is not present on output might be confusing and has to do with how commands are parsed. The best way to explain it is with an example (see "Querying Lists" below).

In dcecp, there are several data structures used to represent DCE objects, such as attributes, bindings and ACLs. Most of these had some representation in the existing control programs, known as the ***string syntax*** of the data structure. The dcecp supports some of it for backward compatibility. Many data structures used by dcecp are lists, and to make it easier for Tcl to handle them, dcecp uses the ***Tcl syntax***. An example of the string syntax of an ACL is:

```
  user:pascal:-rw----
```

The corresponding Tcl syntax is:

```
  {user pascal -rw----}
```

There are several Tcl commands to create and modify lists as well as to search for an element or get an element of a list.

Querying Lists: The Tcl command `lindex` takes two arguments, a list and an integer (remember, both of these are just strings, and it is the `lindex` command that interprets the string arguments specially). It returns the element indicated by the second argument. Elements in lists are numbered from zero; so the following command will return the second element:

```
dcecp> lindex {a b c} 1
b
```

The braces in the example are used to group the elements of the list so that they are passed as one argument to the lindex command. Now, the following example returns a list:

```
dcecp> lindex {a {b c} d} 1
b c
```

The result is without braces around it, but it is a perfectly valid list. To prove this, pass it to the llength command which takes a list as an argument and returns the number of elements as an integer (in Tcl, square brackets perform command substitution, much like backquotes do in command shells):

```
dcecp> llength [lindex {a {b c} d} 1]
2
```

The lrange command takes a list as its first argument and two indices (start and end) as second and third arguments. It returns the specified range of elements from the list. The third argument can be the keyword end to indicate all elements up to the end of the list. Here are some simple examples:

```
dcecp> llength [directory show /.:]
16
dcecp> lrange [directory show /.:] 7 9
{CDS_InCHName new_dir}
{CDS_DirectoryVersion 3.0}
{CDS_ReplicaState on}
dcecp> lindex [directory show /.:] 8
CDS_DirectoryVersion 3.0
```

Manipulating Lists: The list command takes each argument as a list element and displays the list. The lappend command takes a list variable as its first argument, appends the rest of its arguments to the variable and displays it. If the list variable does not exist, it creates one. The concat command displays a single list made by joining multiple lists. This is shown is the following example:

```
dcecp> set a {1 2 3}
1 2 3
dcecp> set b [list $a 4]    --or--    set b "{$a} 4"
{1 2 3 } 4
dcecp> concat $a $b
1 2 3 {1 2 3 } 4
dcecp> lappend a 4
1 2 3 4
```

The linsert command has the following syntax:

```
    linsert <list> <index> <value> <value> ...
```

It returns a new list formed by inserting all <value> arguments before the <index> element of <list>. The lreplace is used to replace a range of list elements with new elements or to delete a range of elements (if you do not specify new elements); the result is returned as a new list. It has the following syntax:

```
    lreplace <list> <first> <last> <value> <value> ...
```

Neither the linsert nor the lreplace commands stores the result of the operation in the list variable; they just display it. These two commands are illustrated in the following example:

```
dcecp> set a { a {b c} d e}
 a {b c} d e
dcecp> linsert $a 2 ch
 a {b c} ch d e
dcecp> lreplace $a 1 1 b c
 a b c d e
dcecp> lreplace $a 1 1
 a d e
dcecp>
```

The split command displays a list created by splitting a string at specified characters (the default character is the white space). The join command is the inverse of split; it concatenates the elements of a list with a specified character between them and returns a string.

```
dcecp> split ″a:b:c″ :
a b c
dcecp> join {a b c} :
a:b:c
```

Searching and Sorting Lists: The lsearch command returns the index of an element in the list or -1 if not present. Its syntax is:

```
    lsearch [-exact|-glob|-regexp] <list> <pattern>
```

If <pattern> is present in <list>, the *index* of its first appearance is returned. The optional switch modifies the search. Use -exact for exact mapping and -glob for pattern matching similar to that of UNIX shells. The -regexp switch uses pattern matching similar to the one used by egrep command, for example:

```
dcecp> set a {tcl Tcl.1 Tcl.2 -4.5 5 8.9}
tcl Tcl.1 Tcl.2 -4.5 5 8.9
dcecp> lsearch -glob $a T*
1
dcecp> lsearch -regexp $a ^(-)\[0-9[(.\[0-9[*)$
3
```

The lsort command sorts a list in a variety of ways, depending on the optional switches: -increasing or -decreasing for the order and -ascii, -integer or -real for the comparison function. For special purpose sorting needs, you can specify your own comparison function with the -command switch.

```
dcecp> lsort -increasing $a
-4.5 5 8.9 Tcl.1 Tcl.2 tcl
```

9.3.4 Control Flow

The dcecp control program provides several commands for controlling the flow of execution in a script. These control flow commands are similar to those found in the C programming language, such as if, for, while, foreach, and case.

9.3.4.1 The if Statement

The syntax of the if statement is:

```
   if <test1> <true_body1> [[elseif <test2> <true_body2>]* else <false_body> ]
```

The <test1> argument is evaluated and if its value is true, then the <true_body1> executes as a script and returns its value. Tcl considers an expression true when the value returned is different from zero. The <elseif> and <else> parts are optional. If <test1> returns a zero value, then it evaluates <test2> as an expression. If its value is non-zero, it executes <true_body2>. It continues evaluating all the <elseif>s until one succeeds. If no test succeeds, then <false_body> executes as a Tcl script.

9.3.4.2 The switch Statement

The switch statement is a better way of writing a program when a series of if-elses occurs. It has two formats:

```
switch [flags] <string> <pattern1> <body1> <pattern2> <body2> ...
switch [flags] <string> { <pattern1> <body1> <pattern2> <body2> ... }
```

The type of flags are the same as used for the search (see "Searching and Sorting Lists" on page 155). They are -exact, -glob and -regexp. This command matches the <string> against each <pattern> in order until a match is found; then it executes the <body> corresponding to the matching pattern. If the last pattern is <default>, then it matches anything. It returns the result of the body executed or an empty string if no pattern matches.

9.3.4.3 The foreach Loop

When you want to perform a given operation on each element in a list, use the foreach command. The syntax of the foreach command is:

```
foreach <var_name> <list> <body>
```

For each element of <list>, in order, it sets the variable <var_name> to that list element and runs the script <body> as a Tcl script. The foreach command returns an empty string.

9.3.4.4 The while Loop

The while loop takes two arguments: an expression and a script. When the <expression> evaluates to non-zero, the while command executes the <body> and reevaluates the <expression>, continuing the loop until the <expression> evaluates to 0. The syntax is:

```
while <expression> <body>
```

9.3.4.5 The for Loop

The for loop behaves like the for loop in C. The syntax is:

```
for <init> <test> <reinit> <body>
```

It executes <init> as a Tcl script. Then it evaluates <test> as an expression. If it evaluates to true, it executes the <body> script, the <reinit> script, which is normally used to increase counter values for the next test, and eventually evaluates <test> again. The loop repeats until the test becomes false.

9.3.4.6 The continue and break Statements

The continue and break commands terminate loops started with the while, for and foreach commands. The continue command terminates the current iteration of the innermost looping command and goes on to the next iteration of the command. The break command terminates the innermost loop execution.

9.3.5 Procedures

A procedure in Tcl can perform a particular task and supports the concept of modular programming design techniques. You can define procedures with the proc command. Its syntax is:

```
proc <name> <parameters> <body>
```

The <name> argument defines the name of the procedure to be created. The second argument stands for a list of parameters used by the procedure. The third argument is the implementation of the procedure. Once defined, it can be used like any other command. The result of a procedure is the result returned by the last command in the <body> script. You can invoke the return command if you want a procedure to return early without executing its entire script.

Many dcecp commands return a list of lists (in the form of an attribute-value list) as a result of their execution. Sometimes, we just need one attribute from that list; so we can define a procedure to obtain a specific attribute from that list:

```
dcecp> proc get_attr {list pattern} {
>  foreach i $list {
>      if { [lsearch -regexp $i $pattern] >= 0 } {
>         return $i
>         }
>  }}
dcecp> get_attr [acl show /.:] unauth
unauthenticated r--t---
dcecp> get_attr [dir show /.:] CDS_ReplicaSt
CDS_ReplicaState on
```

The get_attr procedure takes two arguments; the first one is the list of lists and the second argument is the pattern we are looking for. In a for loop, one element in the list after another is stored in the i variable, and lsearch is called with the current i. We use a regular expression type search so that we can find the pattern even if the name is not completely given. When the pattern is found, its value is returned.

Inside the procedure, all variables have a local scope, and they are undefined after the procedure terminates. If a procedure needs to reference a global variable, it has to use the global command. A local variable can override or hide a global variable by choosing the same name. This is illustrated in the following example:

```
dcecp> set a 5
5
dcecp> proc a_print {} {puts $a }
dcecp> a_print
Error: can't read "a": no such variable
dcecp> proc print_a {} { global a; puts $a}
dcecp> print_a
5
```

The upvar command provides a mechanism for accessing variables outside the context of a procedure. Instead of a value, the argument passed to the procedure is a variable *name*, basically a pointer to the variable's *value*. It is similar to the call-by-reference argument passing. Its syntax is:

```
upvar [level] <var_name> <local_var_name>
```

The level argument is optional and indicates how many levels up the call stack you are referencing. You can specify an absolute number with a #number syntax. The global level is #0. The second argument is the name of the variable you want to access, and the third one is its local name. The following example illustrates the difference between the argument's value (variable name *b*) and the value it indirectly references (5):

```
dcecp> proc print_var { var_name } {
>   upvar $var_name x
>   puts ″$var_name = $x ″
> }
dcecp> set b 5
5
dcecp> print_var b
b = 5
```

9.3.6 Files

Tcl provides commands for reading and writing files. The commands are similar to the procedures in the C standard I/O library. You use the same UNIX-style file names to reference a file.

9.3.6.1 Opening and Closing Files

You can open a file for reading and writing using the open command. The first argument to open is the name of the file; the second argument specifies the file access mode. The access mode may have one of the following values:

r	Reading only. The file must already exist.
r+	Reading and writing. The file must already exist.
w	Writing only. Truncate the file if it already exists; otherwise create a new one.
w+	Reading and writing only. Truncate the file if it already exists; otherwise create a new one.
a	Writing only. Set the initial access position to the end of the file. If the file doesn′t exist, create a new one.
a+	Reading and writing. Set the initial access position to the end of the file. If the file doesn′t exist, create a new one.

Tcl also supports the specification of the file access mode as a list of POSIX flags, such as RDONLY and CREAT. The open command assigns a file identifier to each file when it is opened:

```
dcecp> open /etc/passwd r
file9
```

Use the file identifier to refer to files in subsequent commands. There are three file identifiers already defined and always available for use: stdin (standard input), stdout (standard output) and stderr (standard error). To close a file, use the close command with the file identifier as argument:

```
dcecp> close file9
```

9.3.6.2 Reading and Writing Files

To read a new line from a file, use the gets command:

```
    gets <file_id> [<var_name>]
```

The first argument is the file identifier on which the read is going to be performed. The second argument is optional and refers to the name of a variable where you want to store the line. If it is specified, the command places the line in that variable and returns a count of characters in the line (or -1 for end of file). If it is not specified, it returns the line read as a result (or empty string on end-of-file). The puts command has the following syntax:

```
    puts [-nonewline] <file_id> <string>
```

It writes <string> to the file referenced by <file_id>, appending a newline character unless -nonewline is specified. If no <file_id> is specified, it defaults to standard output. Examples for this are:

```
dcecp> open /etc/passwd r
file9
dcecp> gets file9 line
22
dcecp> puts $line
root:!:0:0::/:/bin/ksh
```

To write out any buffered output that has been generated for a file, use the flush command with the file identifier as argument. To test if an end-of-file condition has occurred, use the eof command on the file identifier.

9.3.6.3 Random File Access

File I/O is sequential by default. To access the files non-sequentially, you can use the seek, tell and eof commands. To change the current access position, use the seek command:

```
seek <file_id> <offset> [start|current|end]
```

This command changes the current access position of the file identified by <file_id>; so the next access starts at <offset> (a positive or negative number) bytes from the origin. The origin is the last argument. The default for it is start. The tell command returns the current access position of a particular identifier.

9.3.6.4 File Management

To manipulate file names, Tcl provides two commands: glob and file. The glob command takes one or more patterns as arguments and returns a list of all the file names that match the pattern:

```
dcecp> cd /usr/include/dce
dcecp> glob pt*.h
pthread.h pthread_exc.h
```

The file command has many options to manipulate file names and to retrieve information about files. Some examples:

```
dcecp> file dirname /a/b/c
/a/b
dcecp> file extension /a/b/program.c
.c
dcecp> file tail /a/b/c.bak.c
c.bak.c
dcecp> file type /etc
directory
dcecp> file type /usr/bin/ls
file
dcecp> file type /dev/rootvg
characterSpecial
```

It also has an option to list the last access and modification time (atime, mtime), to test if the file exists (exist) and whether it is an executable (executable), a directory (isdirectory) or a file (isfile).

The stat option invokes a stat() system call and returns the information on an array. The following example illustrates this (some comments are added to clarify the meaning of each value):

```
dcecp> file stat /etc/passwd passwd_stat
dcecp> foreach i [array names passwd_stat] {
>   puts [format "%-6s is %s " $i $passwd_stat($i)]
> }
mtime  is 802989564       # last modification time
atime  is 804542794       # Time of last access
gid    is 7               # group id
nlink  is 1               # Number of links
mode   is 33204           # Mode bits for file
type   is file            #
ctime  is 802989564       # Time of last status change
uid    is 0               # Owner's user id
ino    is 454             # Inode number
size   is 333             #
dev    is 655365          # Identifier for device containing file
```

9.3.6.5 File I/O Example

We can write a script that prints the name of the users that are defined in both the /etc/passwd file and the DCE Security Registry:

```
dcecp> set principals [principal catalog]
/.../cell1.itsc.austin.ibm.com/nobody
/.../cell1.itsc.austin.ibm.com/root
/.../cell1.itsc.austin.ibm.com/sys
 .
 .
/.../cell1.itsc.austin.ibm.com/hosts/sys5/self
/.../cell1.itsc.austin.ibm.com/hosts/sys5/cds-server
dcecp> set fd [open /etc/passwd r]
file8
dcecp> while { [gets $fd line_fd] >= 0 } {
> set user_name [lindex [split $line_fd :] 0]
> if [lsearch -regexp $principals $user_name] {puts stdout $user_name}
> }
root
bin
sys
adm
uucp
guest
lpd
dcecp> close $fd
```

First, we store the list of the cell's principals in the principals variable. We open /etc/passwd for reading and store the file identifier in the fd variable. The while loop reads all lines of the passwd file, and for each line, it extracts the user's name, checks whether the user is also a DCE principal and, if so, prints the name.

9.3.7 Executing External (Operating System) Commands

Although dcecp is versatile, there are times when you may want your script to use operating-system commands to accomplish some operation. The exec command provides a way for scripts to perform Tcl-external commands. It does that by forking a subprocess in which the command executes. The exec command supports I/O redirection of standard input or output similar to the one provided by UNIX shells (<, <<, >> and |). You can use the ampersand (&) to run a subprocess in background. Some examples are:

```
dcecp> exec grep guest /etc/passwd
guest:!:100:100::/home/guest:
dcecp> exec grep guest /etc/passwd | awk -F: "{print \$3}"
100
```

In the last example, we must put the backlash to prevent Tcl from interpreting $3 as a variable. It is important to consider that exec does not perform any file name expansion. This has to be done with the glob command:

```
dcecp> exec rm *.c
Error: rm: *.c: A file or directory in the path name does not exist.
dcecp> exec rm [glob *.c]
Error: rm: test1.c test2.c: A file or directory in the path name does not exist.
dcecp> eval exec rm [glob *.c]
```

The first remove fails because the exec does not perform file expansion; glob does, but since it returns its result as a single string, and the operating system could not find a file with "test1.c test2.c" as name, it also fails. The solution is to use the eval command to reparse the output from the glob command so that it gets divided into multiple words.

The open command can also be used to create a subprocess and attach its input or output to a pipeline that can be used with the gets and puts commands. To define a pipe, the first character on the file name given to the open command must be the pipe character (|) as the following example shows:

```
dcecp> set f1 [open "|ls " r ]
file6
dcecp> gets $f1
acct.h
dcecp> gets $f1
acct.idl
dcecp> pid $f1
26556
```

The pid command can be used to get a list of the process identifiers in the pipeline associated with an open file.

9.3.8 Other Tcl Commands

Tcl provides a command to obtain information about the state of the interpreter: the info command. This command can take several options. For example, the exists option allows us to query for the existence of a variable:

```
dcecp> set CDS_CH ev2_ch
ev2_ch
dcecp> info exists CDS_CH
1
dcecp> info exists Undefined_var
0
```

There are other options that operate on variables: vars returns the names of all variables accessible at the current level of procedure call, globals returns the names of global variables and locals returns the names of local variables.

Other options available with the info command return information on procedures. The procs option returns a list of all defined procedures. The args option returns a list of the argument names of a procedure:

```
dcecp> info args get_attr
list pattern
dcecp> info body get_attr
foreach i $list {
  if { [lsearch -regexp $i $pattern] >= 0 } {
 return $i
     }
}
```

The body option returns the procedure′s body. The commands option returns a list of all available commands.

Tcl provides the time command to measure the performance of Tcl scripts. It takes two arguments: a script name and a repetition count. It returns the execution average time of each repetition:

```
dcecp> time {principal show cell_admin} 5
85879 microseconds per iteration
```

The rename command changes the name of a command; it takes two arguments: current name and new name of the procedure. If the new name is an empty string, the command is deleted.

9.4 The DCE Control Program

This section explains the dcecp command syntax, user interface, customization, and error handling. Tcl programming constructs needed to extend the dcecp command or write dcecp scripts are discussed 9.3, "Tcl Language Components" on page 149.

9.4.1 DCE Command Syntax

The dcecp commands are implemented as procedures in a Tcl interpreter. Commands return strings, and the main interpreter loop displays the results to the user. Errors are displayed as well. In most cases, a Tcl object-oriented approach has been used to define the commands. The syntax for dcecp commands is the following:

```
    <object> <operation> <name> [-<option> [<value>]]...
```

In this approach, a command is represented by a type of object, and its first argument, the operation, actually defines what needs to be performed on the particular instance (<name>) of the object. Operation names are consistent whenever appropriate. For example, create is always used to create a new instance of the object; delete is always be used to delete an instance. The third and following arguments are options and values. Some commands might require one or more options. Options have a leading dash (-) and are a full word rather than a single letter, but may be abbreviated (see 9.4.2.4, "Abbreviations" on page 167 for details). Some options take a value which immediately follows the option. All arguments are strings, and all commands return strings as values.

The commands available to the user are those known by the Tcl interpreter. In the dcecp, the interpreter knows about the Tcl built-in commands and the additional dcecp commands.

If a command name that Tcl does not know about is invoked, then Tcl calls the Tcl built-in command unknown with the entered command and arguments. The user is free to define this command as desired to have a variety of things happen. By default, Tcl defines the unknown command to use the following heuristic:

1. See if the Tcl autoload facility can locate the command in a Tcl script file. If so, load it, and execute it.
2. If running interactively, see whether the command exists as an executable UNIX program. If so, exec the command.
3. If the command was invoked at the top-level:
   - See if the command requests csh-like history substitution in one of the common forms, such as !!, !<number>, or ^<old>^<new>. If so, emulate csh's history substitution.
   - See if the command is a unique abbreviation for another command. If so, invoke the command.
4. Otherwise, fail with an error indicating an unrecognized command.

This means that from the dcecp> prompt, for instance, a UNIX user can type ls to see a listing of the files in the current directory or rm foo to delete the file foo.

9.4.1.1 Object-Operation vs. Operation-Object

The choice of using an object-operation command order has been controversial. Many parties have stated that an operation-object ordering is more intuitive and is more common in other user interfaces. The major advantage to using an object-operation model is extensibility. To add a new object type to dcecp, that object type merely needs to be added with all supported operations. If an operation-object order were used, then each operation that was supported by the object would need to be taught about the new object. This would be a big problem as operations are expected to be common among objects, such as, create, delete and show.

In deference to the operation-object faction, dcecp includes a script that can be loaded by any user or configured to be loaded for an entire system. This script defines commands named for operations, such as create, delete and show. The commands expect their first argument to be the name of an object and will try to execute the operation on that object using the object-operation syntax. This provides an operation-object syntax to the user in addition to the object-operation order. The commands are based on the Tcl info command and the operations command supported by each object. Therefore, as new objects are added to the system, these commands will learn about them automatically. A sample (and incomplete) implementation of the show command is shown below:

```
proc show {obj args} {
  if {[llength [info commands $obj*]] < 1} {
    error "Object $obj does not exist"
  } elseif {[llength [info commands $obj*]] > 1} {
    error "$obj not unique"
  } else {
    if  {[llength [lsearch [$obj operations] show]] < 1} {
      error "show command not supported by $obj object"
    } else {
      $obj show $args
    }
  }
}
```

Note that the user of these commands will not have to understand the above script. Since the difference is just transposing the first two words, it should not be a problem for a DCE administrator to use the operation-object syntax.

9.4.1.2 Attribute Lists and Options

Many of the commands need to specify attributes to operate upon. For example, the `modify` command allows attributes to be changed, and the `create` command often allows attributes to be created along with the object. In all cases, a mechanism exists so that an attribute list is used to specify the attributes and their values. This makes passing information from one command to another very easy. For example, an ACL copy operation could be written as follows:

```
# copy acl name1 to acl name2
# no error checking
proc acl_copy {name1 name2} {
  acl replace $name2 -acl [acl show $name1]
}
```

While attribute lists are useful for writing scripts, they are often not user-friendly. For those objects that have a fixed selection of attributes, such as *principal* and *dts*, there is an option for each attribute that takes a single value. So, for example, the following two commands are equivalent:

```
principal create nordpol -attribute {{quota 5} {uid 123}}
principal create nordpol -quota 5 -uid 123
```

9.4.1.3 Lists of lists

Since Tcl lists are confusing enough, it′s worth noting here a common usage when lists contain lists. See the 9.3.3, “Data Structures” on page 151 section above for a brief description of Tcl lists. The following example is the command to remove some ACLs from the object `/.:/foo`:

```
dcecp> acl modify /.:/foo -remove {user nordpol}
```

The argument to the `-remove` option is an ACL entry. The ACL entry happens to be a list where the first element describes the ACL type, in this case `user`, and the second is the key for which user, in this case `nordpol`. However, the `-remove` option may take a list of ACL entries; so the following is valid as well:

```
dcecp> acl modify /.:/foo -remove {{user nordpol} {user sally}}
```

All of this seems pretty clear. The point to remember is that lists of one value that do not contain spaces do not need braces. The string syntax of an ACL entry allows the type and key to be separated by a colon (**:**). So, all the following examples are valid:

```
dcecp> acl modify /.:/foo -remove user:nordpol
dcecp> acl modify /.:/foo -remove {user:nordpol user:sally}
dcecp> acl modify /.:/foo -remove {{user nordpol}}
dcecp> acl modify /.:/foo -remove {user:nordpol}
dcecp> acl modify /.:/foo -remove {{user:nordpol} {user:sally}}
```

The following is not legal since it is a list of a list of a list, and the value of the `-remove` option is a list of lists:

```
acl modify /.:/foo -remove {{{user nordpol}}}
```

Also note that lists are just strings with spaces and that braces are just one way to quote the spaces from being interpreted as argument separators. Double quotes and backslashes would work as well. The following are all valid:

```
acl modify /.:/foo -remove {"user nordpol" "user sally"}
acl modify /.:/foo -remove user:nordpol\ user:sally
```

9.4.2 User Interface

The dcecp presents a command-line interface to the user with command-line recall and editing capabilities. The commands and command-line editing interfaces are described below.

9.4.2.1 Invocation

The dcecp is a command-line user interface for administrative commands. There are several methods of invocation:

1. The user starts dcecp and then sees the dcecp prompt:

   ```
   # dcecp
   dcecp>
   ```

   The above is the default prompt. It can be changed using the standard Tcl tcl_prompt1 and tcl_prompt2 mechanisms.

2. The user invokes dcecp with one argument which is a filename of a dcecp script. The script is run, and then dcecp exits. This method of invocation allows interpreter files (those with #!/bin/dcecp as the first line) to work.

   Tcl sets the argc, argv, and arg0 variables to provide access to command-line arguments.

3. The user invokes dcecp with the -c option followed by a set of commands, all on one line separated by semi-colons (;) which must be quoted in the shell. The commands are executed, and then dcecp terminates. For example:

   ```
   # dcecp -c "directory create /.:/foo"
   ```

   The dcecp program also accepts a -s option which causes it not to contact any servers during initialization. Specifically, it will not try to inherit a login context from the invoking process. This can be useful when trying to invoke dcecp when DCE is not functioning properly.

9.4.2.2 Initialization

When dcecp is invoked, it executes the following scripts in the order shown:

1. [info library]/init.tcl — Contains standard Tcl initialization commands on a per-host basis as well as definitions for the unknown command and the auto_load facility. Administrators should avoid adding dcecp customizations to this file. The info library command usually resolves to /opt/dcelocal/tcl.

2. /opt/dcelocal/dcecp/init.dcecp — Contains dcecp-specific startup information for the host. This affects all instances of dcecp running on a host. The dcecp scripts implementing commands and tasks are stored in the /opt/dcelocal/dcecp directory.

3. $env(HOME)/.dcecprc — This file stores user customizations.

9.4.2.3 Command-Line Editing Commands

Previously entered commands can be retrieved. Retrieved commands or manually typed-in lines may be edited before being sent to the dcecp by typing control characters or escape sequences.

Most editing commands may be given a repeat count, <n>, where <n> is a number. To enter a repeat count, type the escape key, the number and then the command to execute. For example, ESC 4 Ctrl-f moves forward four characters, or ESC 4 ESC f moves forward four words. If a command may be given a repeat count, then the text [n] is given at the end of its description.

The following control characters are accepted (not case-sensitive):

Ctrl-A	Move to the beginning of the line
Ctrl-B	Move left (backwards) [n]
Ctrl-D	Delete character [n]
Ctrl-E	Move to end of line
Ctrl-F	Move right (forwards) [n]
Ctrl-G	Ring the bell
Ctrl-H	Delete character before cursor (backspace key) [n]
Ctrl-I	Complete filename (tab key); see below
Ctrl-J	Done with line (return key)
Ctrl-K	Kill to end of line (or column [n])
Ctrl-L	Redisplay line
Ctrl-M	Done with line (alternate return key)
Ctrl-N	Get next line from history [n]
Ctrl-P	Get previous line from history [n]
Ctrl-R	Search backward (forward if [n]) through history for text; must start line if text begins with an uparrow
Ctrl-T	Transpose characters
Ctrl-V	Insert next character, even if it is an edit command
Ctrl-W	Wipe to the mark
Ctrl-XCtrl-X	Exchange current location and mark
Ctrl-Y	Yank back last killed text
Ctrl-[	Start an escape sequence (escape key)
Ctrl-]*c*	Move forward to next character *c*
Ctrl-?	Delete character before cursor (delete key) [n]

The following escape sequences are provided (case-sensitive):

ESC Ctrl-H	Delete previous word (backspace key) [n]
ESC DEL	Delete previous word (delete key) [n]
ESC SPC	Set the mark (space key); see Ctrl-XCtrl-X and Ctrl-Y above
ESC .	Get the last (or [n]'th) word from previous line
ESC ?	Show possible completions; see below
ESC <	Move to start of history
ESC >	Move to end of history
ESC b	Move backward a word [n]
ESC d	Delete word under cursor [n]
ESC f	Move forward a word [n]
ESC l	Make word lowercase [n]
ESC u	Make word uppercase [n]
ESC y	Yank back last killed text
ESC w	Make area up to mark yankable
ESC *nn*	Set repeat count to the number *nn*

There is filename completion. Suppose the current directory has the following files in it:

```
bin    vmunix
core   vmunix_old
```

The following example illustrates the use of file name completion feature (type in what is marked bold):

```
dcecp> rm vm<TAB>
dcecp> rm vmunix<ESC>?
vmunix vmunix_old
dcecp> rm vmunix_<TAB>
dcecp> rm vmunix_old
```

This is actually all happening on the same line. If you start typing a file name and then press the tab key, as much of the name as possible will be finished off by adding unix. Because the name is not unique, it will then beep. If you type the escape key and a question mark, it will display the two choices. If you then continue typing the next character (the underscore) and press the tab key, the file name will be completed.

9.4.2.4 Abbreviations

The dcecp makes use of two mechanisms to allow all object names, operation names and options to be abbreviated to the shortest unique string. The first is a standard Tcl mechanism built-in to the Tcl unknown command. However, this mechanism only works if the command is interactively entered. If the command is found in a script, abbreviation checking is not performed by the standard implementation of unknown. This is to discourage the practice of using abbreviations of commands in scripts.

The other mechanism used for abbreviations is built-in to the individual dcecp commands themselves. They all share the same parser code which is used for both the operation names and the option names. This allows operations and options to be abbreviated to the shortest unique string representing a supported operation or option. For example, in most cases -member can be abbreviated -m since few other options begin with ″m″. Note that this form of abbreviation is always available, whether invoked interactively or via a script.

9.4.3 Help Facilities

The dcecp command offers help in several ways. To see a list of objects provided by the DCE control program, enter the help command. All objects support the operations command to list the operations supported on them. For example, to list the operations that can be performed on the principals, enter:

```
dcecp> principal operations
catalog create delete modify rename show help operations
```

To get more detailed help about an object and its operations, type <object> help. For example:

```
dcecp> rpcentry help
create              Creates a list of empty RPC Entries.
delete              Deletes a list of RPC Entries.
export              Stores bindings in a list of RPC Entries.
import              Returns the bindings from a list of RPC Entries.
show                Returns the attributes of a list of RPC Entries.
unexport            Deletes bindings from a list of RPC Entries.
help                Prints a summary of command-line options.
operations          Returns the valid operations for command.
```

To get information about available command options, you can call help for a specific operation as shown in the following example:

```
dcecp> rpcentry help export
-binding            A list of string bindings.
-interface          Specify a single Interface ID.
-object             A list of object UUID's.
```

You can use the option -verbose to get help about an object itself.

9.4.4 Convenience Variables

All dcecp commands set several variables on execution to store such information as, for instance, the name of the object operated on, the return value of the last command or the DCE name of the user. Users can substitute the value of these variables into the next command to save typing. These are regular Tcl variables; so any mechanism to perform the variable substitution is supported. The most common method is to prefix a $ before the name of the variable, though the set command can be used as well.

All convenience variables are named with two characters: a leading underscore and a single letter. Currently all are lowercase; however, it should be noted that Tcl is case sensitive. Also, one of the variables is an array; so a subscript must be given with it. Some of these variables are read-only to the user, which is explicitly mentioned below.

The following variables are defined in dcecp:

_b Name of the server bound to for the last command. This is actually a Tcl array where the indexes are used to identify the service. Currently, there is only one defined index: sec.

The value specifies the name of a server in whatever manner the service finds useful. This could be the name of an RPC server entry in the namespace, a string binding or the name of a cell. This variable may not be set by the user.

_c Name of the cell that the principal in the current login context is registered in. See the _u convenience variable below. This variable is read-only; setting it via set will generate an error.

_h DCE name of the current host. This variable is read-only.

_n List of the names entered to the last object command as the <name> argument. These names are the names that the command operated on, typically entered as the third argument.

_o Object type used in the last operation. This variable is read-only.

_p Parent of _n. If _n is a list, then this is a list of the same length. This is accomplished syntactically by removing the last name in the pathname of each element in _n. This variable is read-only.

_r Return value of the last executed command. This variable is read-only.

_s Name of a server to bind to for the next command. This is actually a Tcl array where the indexes are used to identify the service. The currently defined indexes are: sec, cds, dts, and aud.

The value specifies the name of a server in whatever manner the service finds useful. This could be the name of an RPC server entry in the namespace, a string binding or the name of a cell. This variable may be set by the user; it is not set by dcecp.

The values of this variable (array) are treated differently by each service:

- The Security Service uses this variable as a default for the next registry operation. If bound to a read-only replica and an update is requested, dcecp will try to bind to the master registry to perform the change.
- The Cell Directory Service uses this variable in the same way as the cdscp concept of a preferred clearinghouse. If set, CDS will only attempt to communicate with the specified server
- The Auditing Service uses this variable in a similar manner to the CDS server. To contact an audit daemon on another host, set this variable to identify that server.
- The Distributed Time Service behaves the same way.

_u Current simple principal name. The cell name of _u is stored in _c; so the fully qualified principal name is $_c/$_u. This variable is read-only.

9.4.5 Error handling

All commands in dcecp return either a list of some information or an empty string on success. When an error occurs, it generally causes all active Tcl commands to be aborted. To prevent early termination, Tcl provides facilities for scripts to catch errors and invoke error handlers.

You can generate an error by executing the error command; for example, the following could be part of a script:

```
set dts_cat_out [dts catalog]
if { [llength $dts_cat_out] == 0 } {
   error "Unable to find any DTS servers"
  }
```

The catch command lets you trap and ignore errors so your script can continue processing. The argument to catch is a Tcl script, which is evaluated. If the script completes normally, then catch returns 0. If an error occurs in the script, catch traps the error and returns 1 to indicate an error. The command can also take a second argument, which is the name of a variable that catch will modify to hold either the script's return value (if it returns normally) or the error message (if the script generates an error). For example:

```
dcecp> proc test1 {x} {
>  if { $x < 5 } { error "$x is less than 5" }
> }
dcecp> catch {test1 4} msg
1
dcecp> set msg
4 is less than 5
```

The dcecp makes use of Tcl's native error handling facilities. There are two global Tcl variables visible in dcecp, which are *errorInfo* and *errorCode*. The former contains the stack-trace of the possibly nested error messages stored by Tcl, and the latter is meant to be machine-readable information.

On error, dcecp commands will return the message string of the error, raise the Tcl Error exception (which can be caught with the Tcl catch command) and set *errorCode* as appropriate to a list where the first element is DCE. The second element is the numeric value of the status code. The error text is printed to standard error. If the global dcecp_verbose_errors variable is set to one, dcecp will instead output the entire contents of *errorInfo* with the string *"Error:"* prepended. A dcecp script which traps an error is shown below:

```
    # This routine returns 1 if the argument is the name
    # of a CDS directory, 0 if it is not.  It uses the show
    # operation to determine if it is a directory.  If the
    # command works then it is a valid directory, if not,
    # an error is generated and caught by the Tcl catch
    # command and the routine returns 0.

    proc isdirectory {name} {
      if {[catch {directory show $name}]==0} {return 1} {return 0}
    }
```

In cases where an argument list is given to a command indicating that an operation is to be performed on more than one object, the operations are usually performed iteratively. If there is an error, the command will abort at the time of error with the Tcl TCL_ERROR exception. Some operations will have finished and others will not have. The operations are always performed in the order entered, and the error message should make it clear on which object the command failed.

9.5 Putting it Together: A dcecp Programming Example

Most dcecp commands return their output as a list of lists which is sometimes difficult to read, whereas the output of the previous DCE commands were more readable. For example, the output of a ***rpccp*** command may be as follows:

```
# rpccp show entry /.:/servers/PrintServer

objects:

  96b316fe-9c66-11ce-ae77-10005aa86e2d
  3b98b9f0-b551-11ce-aba2-10005a4f4629
  5db11410-b551-11ce-aba2-10005a4f4629

binding information:

  <interface id>   b367ea90-9cf7-11ce-ba29-10005aa86e2d,1.1
  <string binding> ncadg_ip_udp:9.3.1.126[]
  <string binding> ncacn_ip_tcp:9.3.1.126[]
  <string binding> ncadg_ip_udp:9.3.1.241[]
  <string binding> ncacn_ip_tcp:9.3.1.241[]

  <interface id>   5db0f098-b551-11ce-aba2-10005a4f4629,2.1
  <string binding> ncadg_ip_udp:9.3.1.241[]
  <string binding> ncacn_ip_tcp:9.3.1.241[]
```

The corresponding dcecp command produces the following output:

```
# dcecp
dcecp>  rpcentry show /.:/servers/PrintServer
{b367ea90-9cf7-11ce-ba29-10005aa86e2d 1.1
  {ncacn_ip_tcp 9.3.1.241}
  {ncadg_ip_udp 9.3.1.241}
  {ncacn_ip_tcp 9.3.1.126}
  {ncadg_ip_udp 9.3.1.126}}
{5db0f098-b551-11ce-aba2-10005a4f4629 2.1
  {ncadg_ip_udp 9.3.1.241}
  {ncacn_ip_tcp 9.3.1.241}}
{96b316fe-9c66-11ce-ae77-10005aa86e2d
3b98b9f0-b551-11ce-aba2-10005a4f4629
5db11410-b551-11ce-aba2-10005a4f4629}
```

Thanks to its extensibility and Tcl, the dcecp is powerful enough to build our own commands. Following is a program that reformats the output of dcecp rpcentry:

```
 1  #!/usr/bin/dcecp
 2  proc rpcentry_show {entry args} {
 3
 4    set entry_info [eval rpcentry show $entry $args]
 5    set elements [llength $entry_info]
 6    set binding_info [lrange $entry_info 0 [expr $elements - 2]]
 7    set objects_info [lrange $entry_info [expr $elements - 1] end]
 8
 9    puts ″\nobjects:\n″
10    foreach object $objects_info {
11     puts $object
12    }
13
14    puts ″\nbinding information:″
15    foreach bind_inf $binding_info {
16      set interface_id [lindex $bind_inf 0]
17      set version [lindex $bind_inf 1]
18      puts [format ″\n   <interface id>    %s,%s″ $interface_id $version]
19      foreach object [lrange $bind_inf 2 end] {
20      set protocol  [lrange $object 0 0]
21      set ip        [lrange $object 1 1]
22      set end_point [lrange $object 2 2]
23      puts [format ″   <string_binding>  %s:%s\[%s\]″ $protocol \
24              $ip $end_point]
25      }
26    }
27  }
28
29 eval rpcentry_show $argv
```

Figure 61. Example Tcl Script to Reformat DCE Command Output

The procedure in Figure 61 takes two arguments. The first argument is the name of the entry we want to display, and the second argument, with the special value args, means that the rest of the command line is passed as a whole.

The arguments are passed to the rpcentry show command (see line 4), and the output is stored in the entry_info variable. Since the args variable holds all the arguments as one string, we have to call eval to separate the arguments in words. The rpcentry show command returns a list consisting of two or more lists. Each list, except for the last one, contains one interface definition with its binding information, and the last one is made up of object UUIDs. On lines 5 through 7, we separate these elements, and store them in their corresponding binding_info and objects_info variables.

Lines 9 through 11 write a title and display the object UUIDs within a foreach loop. Line 14 puts the title *binding information*.

The binding_info variable is itself a list of lists. It contains a list of interfaces, each of which contains an interface ID, a version and a list of string bindings. Line 15 through 18 build an outer foreach loop iterating over the interfaces, separating its components into variables and printing the interface ID and version number. Lines 19 through 24 build an inner loop over the string bindings of the interface treated in the outer loop. They separate each string binding into variables for the RPC protocol, network address and endpoint and print them in the rpccp format.

The last line invokes the procedure. To use it, we enter this procedure into a file named rpcentry_show and set the executable permission bit. The following are two examples for the output of this new command:

```
# rpcentry_show /.:/servers/PrintServer

objects:

96b316fe-9c66-11ce-ae77-10005aa86e2d
3b98b9f0-b551-11ce-aba2-10005a4f4629
5db11410-b551-11ce-aba2-10005a4f4629

binding information:

   <interface id>    b367ea90-9cf7-11ce-ba29-10005aa86e2d,1.1
   <string_binding>  ncadg_ip_udp:9.3.1.241[]
   <string_binding>  ncacn_ip_tcp:9.3.1.241[]
   <string_binding>  ncadg_ip_udp:9.3.1.126[]
   <string_binding>  ncacn_ip_tcp:9.3.1.126[]

   <interface id>    5db0f098-b551-11ce-aba2-10005a4f4629,2.1
   <string_binding>  ncacn_ip_tcp:9.3.1.241[]
   <string_binding>  ncadg_ip_udp:9.3.1.241[]
#
#
# rpcentry_show /.:/servers/PrintServer -interface \
    5db0f098-b551-11ce-aba2-10005a4f4629,2.1

objects:

96b316fe-9c66-11ce-ae77-10005aa86e2d
3b98b9f0-b551-11ce-aba2-10005a4f4629
5db11410-b551-11ce-aba2-10005a4f4629

binding information:

   <interface id>    5db0f098-b551-11ce-aba2-10005a4f4629,2.1
   <string_binding>  ncacn_ip_tcp:9.3.1.241[]
   <string_binding>  ncadg_ip_udp:9.3.1.241[]
```

Chapter 10. Remote Procedure Calls

Distributed client/server applications in DCE use remote procedure calls (RPCs) to make function calls (transparently) across a network. Other DCE services also use RPCs; they are also client/server applications. RPC is the basis for DCE.

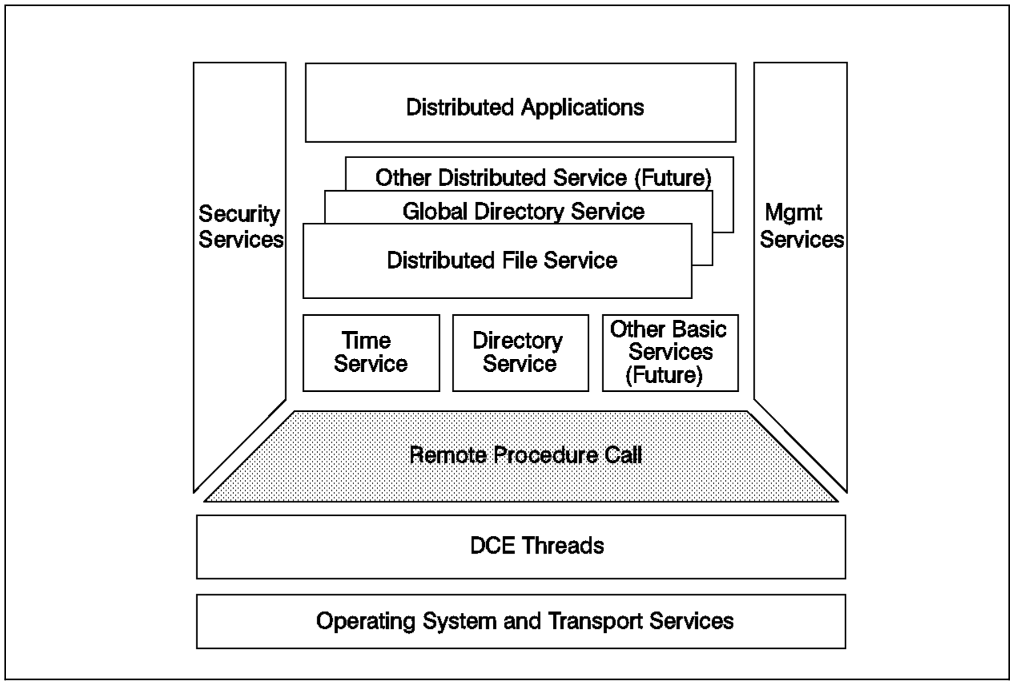


Figure 62. RPC as a DCE Component

This chapter discusses all components involved in the execution of an RPC, including CDS and Security Services access. It explains how clients and servers are written and work together. At the end of the chapter, we give a complete picture of how everything works together.

A DCE application development guide is about RPC programming and how to use the different DCE core services to make the application globally available and secure.

Although this chapter is fairly detailed and comprehensive, it tries to be high-level and cannot contain every detail of RPC programming. Readers who want to know how RPCs work behind the scenes and interoperate with CDS and Security should read this. They will also find a description of application development steps.

10.1 What is RPC?

The RPC application has two sides: the client side which calls the remote procedure and the server side which executes the procedure in its own address space. Clients and servers can be on different computers linked by communications networks.

A procedure is defined as a closed sequence of instructions that is entered from, and returns control to, an external source. Data values may be passed on both directions along with the flow of control. A procedure call is the invocation of a procedure. A local procedure call and an RPC behave similarly; however, there are semantic differences due to the distributed nature of RPCs.

In a local procedure call, the arguments and results are passed on the process′s stack because the calling and the called procedure share the same address space. A local procedure call commonly uses the call by reference passing mechanism for input/output parameters. Due to the differing address spaces of calling and called procedures, RPCs with input/output parameters have copy-in, copy-out semantics.

While a local procedure call depends on a static relationship between the calling and the called procedure, the RPC paradigm requires a more dynamic behavior. In a local application, this relationship is established by linking the calling and called code. Linking gives the calling code access to the address of each procedure to be called. Enabling a remote procedure call to go to the right procedure requires a similar relationship (called a ***binding***) between the client and the server. A binding is a temporary relationship that depends on a communications link.

The use of remote procedures increases complexity in the application design. Issues such as a remote system crash, security problems, communication links, naming, and binding require the use of new functions that are unnecessary for local procedure calls. DCE′s RPC provides a high-level programming model for distributed application development hiding most of these communication details from the application programmers.

An end-user does not see any RPC at all. The minimal amount of administration involved in RPC can usually be handled by the server-side application code. It is the application programmer who most often comes in contact with the RPC component. Application programmers design and implement how RPC behaves. They have to be aware of such things as how the server is going to be located or whether the application will be multithreaded or not.

10.1.1 IDL, Stubs and RPC Runtime

When developing an RPC application, the application designer must create an RPC interface that is a logical grouping of operations, data types and constants that serves as a unique network contract for a set of remote procedures. This RPC interface is stored in an ***interface definition language (IDL)*** file. An example of a simple IDL file is:

```
[
    uuid(de6a43a8-b6f8-11ce-8223-10005a4f4629),
    version(1.0)
]
interface math
{
  long int add( [in] long int a, [in] long int b);
  long int product( [in] long int a, [in] long int b);
}
```

The *math* interface defines two remote operations (*add* and *product*) that the math server provides. Through this interface, a client can request a server to perform an add() operation and return the result as an integer. DCE RPC development tools have a special IDL compiler that automatically generates client and server stub code from the interface definition file. You find a more detailed description of the IDL and IDL files in 10.5, “Developing an RPC application” on page 194.

The ***stubs*** are needed to manage the communication between the client and the server. Figure 63 below shows the RPC layers involved in the communication between client and server. The client stub takes the call with its arguments from the client, prepares it for transmission and passes it on to the runtime library. The ***RPC runtime*** performs such tasks as controlling communications between clients and servers or finding servers for clients on request. The server stub receives the call through the RPC runtime and calls the desired application procedure. The stubs and runtime library are linked into the client and server executables.

You can find a more detailed description of the RPC components in 10.2, “RPC Components” on page 176.

10.1.2 RPC Data Flow

The following figure illustrates how the RPC call is passed through the different software layers and between client and server:

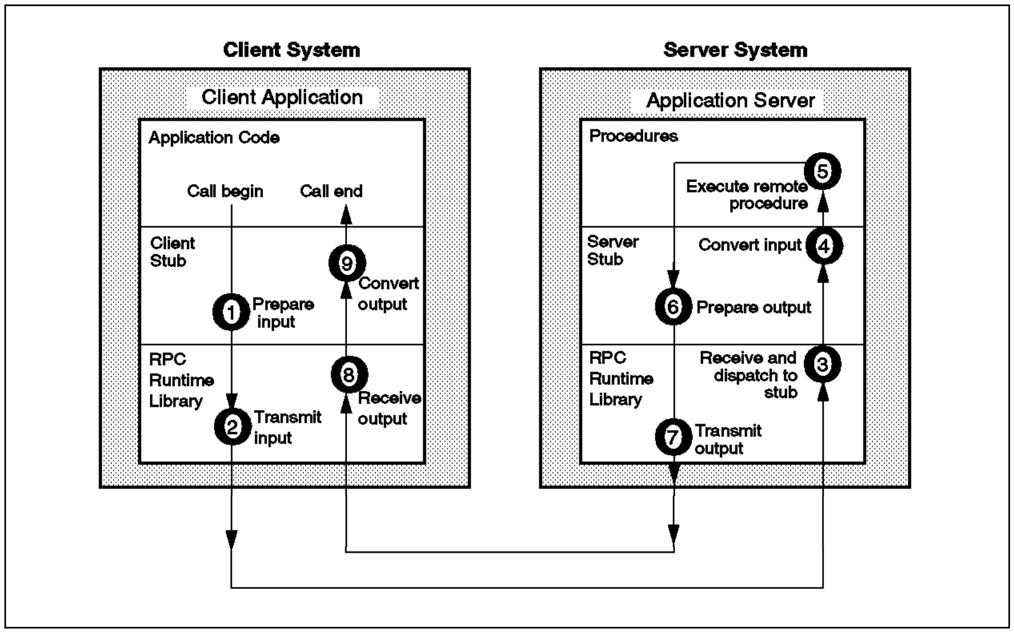


Figure 63. Steps Involved in a Remote Procedure Call

In this figure, the following steps are performed:

1. The client′s application code calls the remote procedure. For the application code, this is like a local procedure call. The client stub gets the input arguments, prepares them for transmission and dispatches the call to the client′s RPC runtime.
2. The client′s RPC runtime transmits the input argument to the server′s RPC runtime.
3. The server′s RPC runtime receives the arguments and dispatches the call to the server stub for the called RPC interface.

4. The server stub converts the input to an appropriate format for the server and calls the procedure.
5. The procedure executes and returns the results to the server stub.
6. The server stub prepares the arguments for transmission and passes them to the server′s RPC runtime.
7. The server′s RPC runtime transmits the results over the communication network to the client′s RPC runtime.
8. The client′s RPC runtime receives the arguments and dispatches them to the client stub.
9. The client stub converts the input to an appropriate format for client and passes them on to the calling procedure.

10.1.3 Synchronous vs. Asynchronous Models

The procedure call model always has a synchronous behavior; that is, the calling procedure must wait until the called procedure terminates. This is true for local and remote procedure calls. The procedure call model is different from other communication models, such as messages queues, that have an asynchronous behavior. Examples of asynchronous models for communications are the Recoverable Queuing Service (RQS) of Encina and the Message Queuing Interface (MQI; part of the IBM MQSeries product).

Asynchronous models are good for batch processing or when the client initiating the request cannot afford to wait for it to complete. Synchronous models have the benefit of assuring the caller that the request was either completed or abnormally terminated when it gets back the results. This type of model is good for applications that require immediate response, such as online transaction processing.

Although RPCs are synchronous, the use of multiple threads of execution allows the client to perform other tasks while one thread is waiting for an RPC to terminate. This does not change the synchronous behavior of RPCs, but gives some of the benefits of asynchronous models to the client application.

10.2 RPC Components

There are several components involved in the processing of an RPC. These components include:

- Stubs
- RPC runtime
- Protocols
- RPC client application
- RPC application server and manager
- Service queues

Other components are needed to support the client in finding and addressing the right application server. They are explained in 10.3, “Finding Remote Services” on page 182. In this section, we will explain the components that process an RPC call.

10.2.1 Stubs

The use of DCE RPCs gives to the application programmer the advantage of thinking in the same way as when he is developing a non-distributed application. The application programmer can use a remote procedure call as if they were calling a local procedure, but, in fact, they are calling the stub code. On the server side of the application, the remote procedure can be implemented just as if a local procedure were calling it. The code that hides all the communication details and provides this view to the programmers is the ***stub code***.

As we mentioned, the stub is the code generated automatically by the IDL compiler from an interface definition file. It is an interface or application-specific code module that provides a basic support function for remote procedure calls. Stubs prepare input and output arguments for transmission between systems with different forms of data representation. The stubs use the RPC runtime to send and receive remote procedure calls. When using *automatic binding* (see 10.3.3, "Binding Methods" on page 187), the client stub can use the RPC runtime to locate a server for the client.

When a client application calls a remote procedure, the client stub first prepares the input and output arguments for transmission. The process of preparing arguments for transmission is known as ***marshalling***. It converts call arguments into byte-stream format and packages them for transmission.

On the server side, upon receiving call arguments, the stub *unmarshalls* them. Unmarshalling is the process by which a stub disassembles incoming network data and converts it into application data by using a format that the local system understands.

Marshalling and unmarshalling permit client and server systems to use a different data representation for equivalent data (for example, the ASCII and EBCDIC character set).

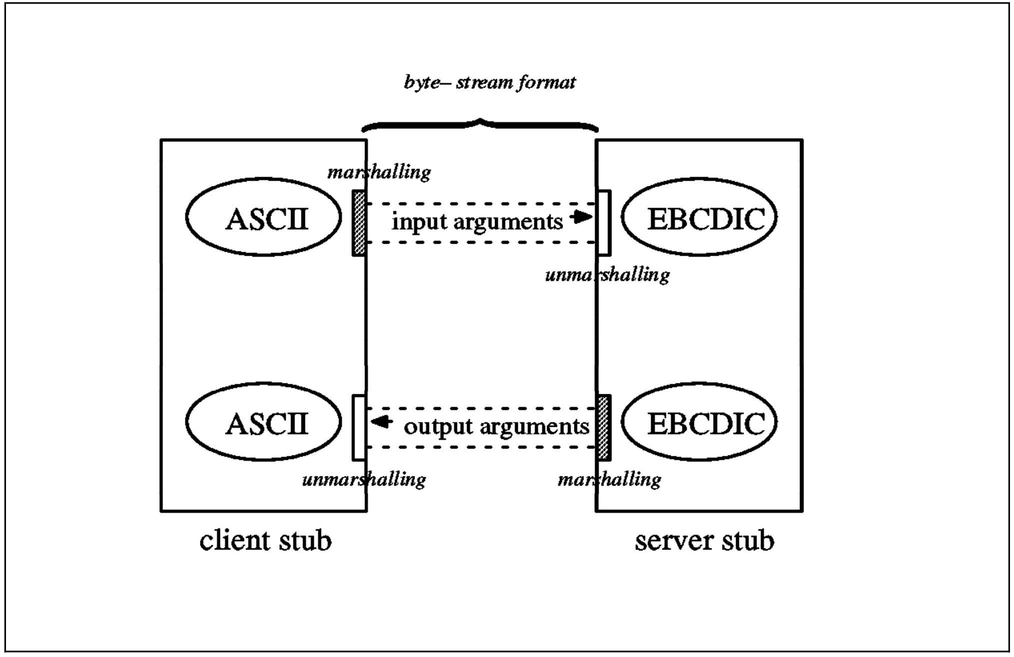


Figure 64. Marshalling and Unmarshalling between ASCII and EBCDIC Data

DCE uses a *receiver-makes-it-right* scheme. All calls are tagged with a description of the calling machine′s basic data representations (for example, how integers, characters and floating-point data are represented). When the call is received, the receiver converts the data from the sender′s to the receiver′s representation, if necessary.

For application-specific types of data, a developer must supply user-defined marshalling routines. The client and server parts of a distributed application are linked with their corresponding stub code to build the executable code.

10.2.2 RPC Runtime

Every RPC client and RPC server is linked with a copy of the RPC runtime. Runtime operations perform such tasks as controlling communication between clients and servers or finding servers for the clients on request. Stubs exchange arguments through their local RPC runtimes.

The client runtime transmits remote procedure calls to the server. The server runtime receives the calls and dispatches each call to the appropriate stub. The server runtime passes the call results to the client runtime.

As shown in Figure 65 below, the DCE RPC runtime provides an API used by the stubs and by user applications. Client application code can use the API for several purposes, such as selecting the type of authentication it wants to use or locating a service. The application server must contain server initialization code with calls to RPC runtime routines when the server is starting up and shutting down.

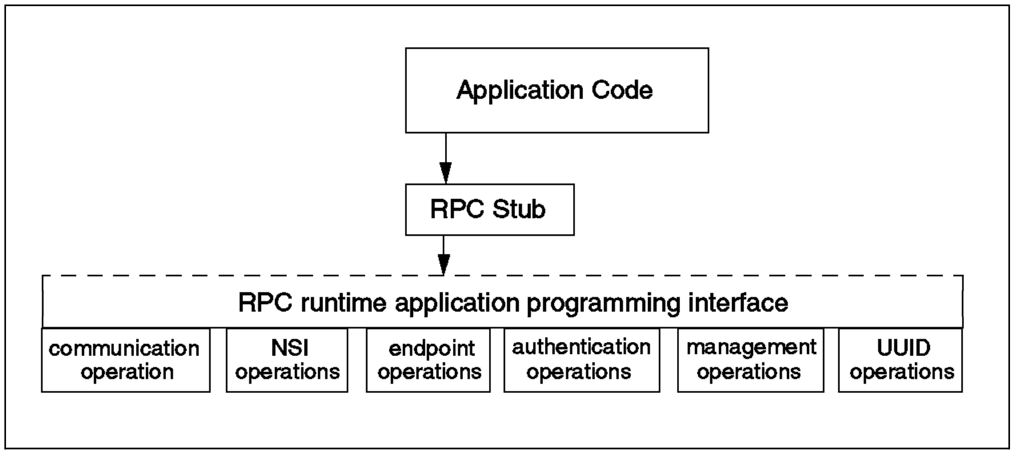


Figure 65. Runtime API Operations

The following list explains the different services provided by the RPC runtime:

- Communication operations

  The RPC runtime is responsible for establishing a binding (the communication link) and for the data transfer between client and server. At initialization, RPC servers makes a number of calls to communications operations, for example, for selecting the protocol sequences to be used.

- Directory service interface operations

  The RPC runtime can be used to store and search for the location of servers (binding information) in the directory service (CDS). The CDS can be accessed through DCE RPC Name Service Interface (NSI). Using the NSI export operation, an RPC server can place information about its interfaces, objects and addresses into a namespace entry. Using NSI import operations, the RPC clients can access this information.

- Endpoint operations

  On one host, there could be several RPC servers running; so a host address is not sufficient to locate a server. The complete address of a server instance is called a fully bound binding handle, and it contains a host address and an endpoint (see 10.3.1, “Binding Handles” on page 182). DCE RPC endpoint operations allow servers to dynamically create their own endpoints in the local endpoint map. Clients can resolve partial binding information into fully bound binding handles that contain the appropriate endpoints.

- Authentication operations

  The authentication operations prove the identity of clients and servers to each other in order to make appropriate authorization decisions. The RPC authentication operations define what authentication mechanism (usually DCE Kerberos) and what protection level will be used for ongoing RPC communication.

- In addition, there are operations to manipulate UUIDs and to manage RPC applications, as well as several other kinds of operations.

10.2.3 Communication Protocols

A communication protocol is a clearly defined set of operation rules and procedures for communications. A communication link depends on a set of communications protocols. The Open Systems Interconnect (OSI) standards body defined a reference model with seven communication layers that depend on each other. Each of these layers covers a specific part of the communication between two computers and uses the lower layer to communicate with its peer on the other system. Each layer defines protocols for doing so.

An example of a protocol set is the Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Internet Protocol (IP), commonly known as TCP/IP. IP implements the network layer of the OSI model. TCP and UDP implement the transport protocol, which both use IP. Other examples of communication protocols are IBM System Network Architecture (SNA), NetBIOS or Novell′s IPX.

An RPC protocol is a communication protocol that supports the semantics of DCE RPC API and is responsible for marshalling and unmarshalling. It runs over specific combinations of transport and network protocols. DCE RPC provides two RPC protocols:

1. Network Computing Architecture Connection-Based Protocol (NCACN)

   This protocol runs over a connection-oriented transport protocol, such as TCP. It guarantees reliability in the delivery of data, and it provides indication of a connection loss.

2. Network Computing Architecture Datagram Protocol (NCADG)

   This connectionless protocol runs over connectionless transport protocols, such as UDP. DG does it ″as best as it can″. Packets are individually addressed. They can follow different network paths; therefore, the sequence of the incoming packets may be mixed up. Packets can get lost. Reliability must be provided by higher layers; it is not protocol-inherent. DG protocol supports broadcast calls.

For two systems to communicate via DCE RPC, their RPC runtimes must support at least one identical communication protocol and transfer syntax. Initially, the supported transport layers were TCP and UDP over IP because they are most widely supported. NetBIOS is supported by DCE for OS/2. The only supported transfer syntax is Network Data Representation (NDR) as used by the NCA.

Furthermore, both AIX and OS/2, support a fast local transport via inter-process communication (IPC) for use by clients and servers running on the same machine. These protocols are called local RPC (LRPC). The LRPC binding handles are not seen outside the machine; they are automatically generated when the RPC components realize that client and server are running on the same machine.

10.2.4 RPC Client Application

The client part of a client/server application is the one that contains the application logic and the user interface. It uses RPC calls to specialized servers which help the client perform the work. Of course, work done by a server upon a single RPC call from a client can itself be quite complex and involve calls to other servers.

DCE RPC programmers have all the flexibility in the world to create client programs. A simple program can use one single server and need not even be aware of the fact that a procedure call is served by remote server. It just has to include a header file created by the IDL compiler and link with the client stub. A complex program can use several different servers offering the same or different interfaces at the same time. In between, there are many variants.

A client has to decide on a ***binding method*** first. If it does not care about which particular server instance it is connected to and the network is relatively small, using the automatic binding method may be sufficient. With this method, a client just calls procedures as if they were local; it does not need to be aware of the fact that it is using RPCs. When a client wants to have control over which server(s) it is going to use, it might want to use implicit or explicit binding, where it can specify the server address. This might be necessary for user-provided load balancing in a large network. For authenticated RPC, it needs access to the binding handle; so it cannot use automatic RPC.

If a client wants to use more than one server in parallel, for instance, to speed up matrix calculations, it needs to be ***multithreaded***. In order to perform RPC in parallel threads, a client must use explicit binding. It is also useful to spawn off a thread for a simple RPC because then you can have a control thread that could kill the RPC thread if anything goes wrong. That is the way to implement a user-controlled timeout mechanism.

10.2.5 RPC Application Server and Manager

The server implements the RPC procedures defined in the RPC interface definition file. A server consist of two parts:

- A server part that initializes the server and defines its capacities
- A manager part that implements the procedures as defined in the IDL

The ***server initialization code*** sets up the environment. Since it is very server-specific, the server initialization needs to be written by the application developer. In several steps, which are explained in more detail in 10.5.6, “Developing a Basic Server” on page 199, the server registers its interfaces, protocol sequences and endpoints with the RPC runtime. This is illustrated in Figure 66 on page 181 as step (1). It then advertises itself into CDS (2) and begins to listen for incoming calls (3).

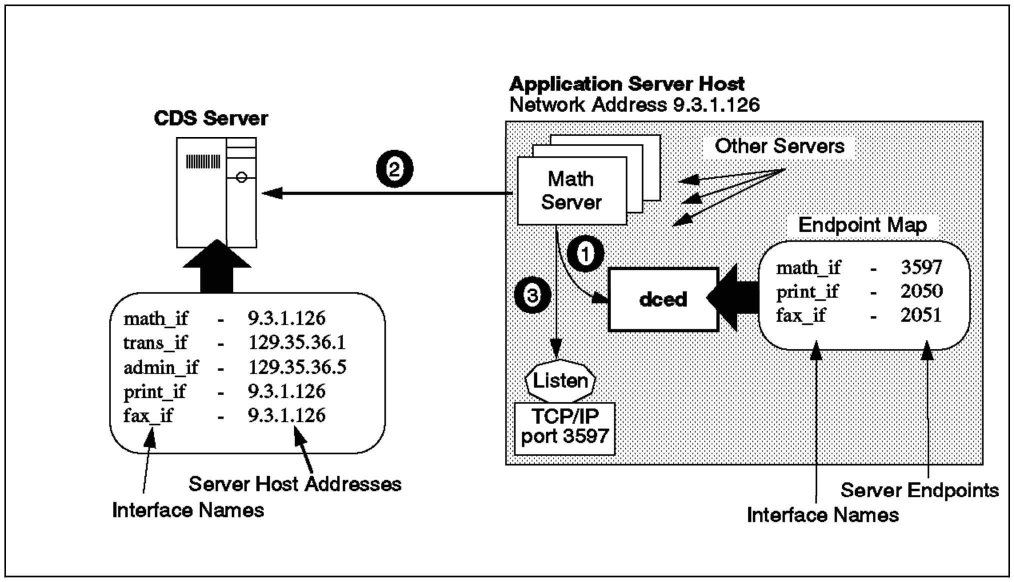


Figure 66. Server Initialization

The server part must also handle the clean-up tasks when it ends. The server should unregister the endpoints from the local endpoint mapper and delete its entries from the namespace before stopping.

The ***manager*** is the set of procedures that implements the RPC interface defined in the interface definition file. For each procedure specified in the IDL file, the manager code must have a corresponding procedure that implements it. A server can have multiple managers for the same interface that can overlay each other. In this case, a type is associated with each manager. The server RPC runtime basically looks at an object UUID coming in with the call and dispatches the call to the right manager. We describe objects and manager types in 10.3.6, “Object UUIDs and Manager Types” on page 189.

A server is always ***multithreaded*** because server part and manager run in different threads. However, if you specify in the `rpc_server_listen()` call that the number of concurrent threads is one, then only one manager thread is created, and it does not have to be thread-safe. As soon as you allow more manager threads, the code must be made thread-safe.

10.2.6 Service Queues

There are two service queues to handle incoming requests in the server:

1. A system request buffer
2. A call queue

When the server declares the protocol sequences it supports to the RPC runtime, it can also specify the maximum number of concurrent remote procedure call requests that the server can accept. At this point, the RPC runtime creates the endpoint(s) and assigns a network buffer for each endpoint that is large enough to accept at least this number of concurrent call requests. These are ***system request buffers***. Each server process regularly dequeues requests, one by one, from all of its request buffers. At this point, the server process recognizes them as incoming calls. An incoming call is rejected if the call queue is full.

As part of the call to initiate listening, the server application specifies the maximum number of concurrent calls it will execute. This number depends on the design of the application. The RPC runtime creates the same number of call threads in the server process. Each call thread can execute one RPC request. Each server process uses a first-in, first-out ***call queue***. When the server is already executing the maximum number of concurrent calls, it uses the queue to hold incoming calls. The capacity of the queues for incoming calls is implementation dependent (in AIX, the queue size is eight times the number of threads).

The appearance of the rejected call depends on the RPC protocol the call is using. If using a connectionless protocol, the call fails as if the server does not exist, returning a *communication failure* status code. If using a connection-oriented protocol, the server rejects the call with a *server too busy* status code.

10.3 Finding Remote Services

The process of finding the server and establishing a relationship over a communication link between the client and server RPC runtimes is called a *binding*. There are several ways in which a client can find a server. The most simple is to hard-code the address, endpoint and protocols of a server into the application. Obviously this implementation is not flexible. A more flexible way is to use the namespace maintained by the Cell Directory Service. A client can find a server by asking the CDS for the location of a server that handles the interface that the client is interested in. This is done using the *Name Service Interface* import operations. A client can also obtain server binding information in string format (called string binding) from an application-specific source, such as a file or an environment variable.

In this section, we describe the components that are important for a client to find a server, which are:

- Binding handles
- Name Service Interface (NSI)
- Binding methods
- DCE daemon and endpoints
- Entry-Point Vector (EPV)
- Object UUIDs and manager types

10.3.1 Binding Handles

In general terms, binding information is data about one or more potential bindings. Binding information includes a set of data that identifies a server to a client or a client to a server. Binding information is maintained in several places:

- Server RPC runtime — Information about the objects, routines and protocols registered by the server as well as information about client(s) currently using this server.
- Server host DCE daemon (endpoint mapper) — A system-wide list with an entry for each combination of supported objects, interfaces, protocols, and endpoints for all application servers.

- CDS — Information about globally available application servers.
- Client RPC runtime — Information about the server(s) it is currently using.

Clients and servers do not have direct access to binding information. They need procedures to obtain a reference, an opaque pointer, to these internal data structures. These pointers are called ***binding handles***. There are other procedures to access and to manipulate binding handles. The binding handles are used by application programs in calls to the RPC runtime or in RPC calls.

```
# rpccp show entry /.:/subsys/dce/sec/master

objects:

  997d0bd2-ce94-11ce-8ab9-10005aa86e2d

binding information:

  .... cut some data

  <interface id>   47b33331-8000-0000-0d00-01dc6c000000,0.0
  <string binding> ncacn_ip_tcp:9.3.1.126[]
  <string binding> ncadg_ip_udp:9.3.1.126[]

  <interface id>   47b33331-8000-0000-0d00-01dc6c000000,1.0
  <string binding> ncadg_ip_udp:9.3.1.126[]
  <string binding> ncacn_ip_tcp:9.3.1.126[]
```

Figure 67. Binding Information in CDS

A good example to illustrate and explain the elements of binding information is the CDS entry in Figure 67. It consists of the following components:

- ***Object UUIDs (optional)***

  A server may manage several distinct objects or resources within one or multiple interfaces. The purpose of object UUIDs is to specify a particular object or resource the server needs to work on.

  What is actually done with the object UUID is completely up to the server implementation. The client does not need to know how the server deals with it; it only has to know when it needs to specify one. In the example in Figure 67, the security server supports one object UUID. For more information on object UUIDs, see 10.3.6, “Object UUIDs and Manager Types” on page 189.

- ***Interface UUID and version number***

  An interface is defined by its interface UUID and a version number. This enables the server to support different versions of the same interface. The version number consists of a major and a minor number. Interfaces with the same major number are considered compatible if the client specifies a minor number that is smaller or equal to the one that is exported by the server.

  The example in Figure 67 shows two (incompatible) interfaces with the same interface UUID but different major version numbers (1.0 and 0.0).

- ***Protocol sequence***

  The protocol sequence specifies the transport and network layer, such as TCP/IP or UDP/IP, and the DCE RPC protocol, such as network computing architecture datagram (NCADG) or network computing architecture connection-oriented (NCACN).

The example in Figure 67 on page 183 shows that the server supports the NCADG protocol over UDP (`ncadg_ip_udp`) and the NCACN protocol over TCP (`ncacn_ip_tcp`).

- ***Network address***

  This is a transport-protocol-dependent address that identifies the host system, for example, the IP address when the TCP/IP protocol is used.

  The example in Figure 67 on page 183 shows that this security server runs on an IP node with address 9.3.1.126.

- ***Endpoint***

  The endpoint is the transport-layer specific address of the process serving the call. In TCP/IP, this is the port number of the socket address the server process is listening for.

  In the above example, Figure 67 on page 183, the endpoint is not defined. CDS stores ***partial binding information*** because the actual endpoint is dynamically assigned by the DCE daemon on that server host. When executing an RPC call with a partly bound handle, the client RPC runtime contacts the remote DCE daemon to obtain the fully bound handle with an endpoint to a compatible server.

Each application server creates this binding information during its initialization with several calls to the RPC runtime. We gave a short introduction on the server initialization in 10.2.5, "RPC Application Server and Manager" on page 180 and give more details on it in 10.5.6, "Developing a Basic Server" on page 199.

Each object UUID (only one in Figure 67 on page 183) is combined with all interface entries and protocols. This yields valid potential binding handles (four in the example). The information exported to the local endpoint mapper looks very similar. However, the endpoint mapper lists every one of these potential combinations as distinct entries and adds the appropriate endpoint to them. By querying the security server machine′s endpoint map, we can see what binding handles are available on the server machine:

```
# rpccp show mapping

  .... cut some data

  <object>         997d0bd2-ce94-11ce-8ab9-10005aa86e2d
  <interface id>   47b33331-8000-0000-0d00-01dc6c000000,0.0
  <string binding> ncacn_ip_tcp:9.3.1.126[2578]
  <annotation>     DCE user registry rdaclif_v0_0_s_ifspec

  <object>         997d0bd2-ce94-11ce-8ab9-10005aa86e2d
  <interface id>   47b33331-8000-0000-0d00-01dc6c000000,0.0
  <string binding> ncadg_ip_udp:9.3.1.126[3428]
  <annotation>     DCE user registry rdaclif_v0_0_s_ifspec

  <object>         997d0bd2-ce94-11ce-8ab9-10005aa86e2d
  <interface id>   47b33331-8000-0000-0d00-01dc6c000000,1.0
  <string binding> ncacn_ip_tcp:9.3.1.126[2578]
  <annotation>     DCE user registry rdaclif_v1_0_s_ifspec
  <object>         997d0bd2-ce94-11ce-8ab9-10005aa86e2d
  <interface id>   47b33331-8000-0000-0d00-01dc6c000000,1.0
  <string binding> ncadg_ip_udp:9.3.1.126[3428]
  <annotation>     DCE user registry rdaclif_v1_0_s_ifspec
```

The new commands for the CDS access and the endpoint mapper listing would be:

```
dcecp> rpcentry show /.:/subsys/dce/sec/master
dcecp> endpoint show
```

When a client wants to connect to a server, it needs to find a ***compatible server***. A server is considered compatible if it offers the same interface UUID, the same major interface version number, the same or a higher minor version number, the same protocol sequence, and the same object UUID as the client requests. The requirement for the *same* object UUID is not so strict. If the client requests a non-nil object UUID not offered by the server, the nil object UUID is considered compatible. On the other hand, if the client requests the nil object UUID, it might get a randomly selected object UUID (including nil) back from CDS.

To establish the actual connection, the client needs a ***server binding handle***. Server binding information is maintained in CDS as well as in the endpoint map and the RPC runtime of the server machine. CDS usually delivers partly bound handles. 10.3.2.2, “Searching The Namespace” on page 186 explains how a client can obtain binding handles from CDS. 10.3.4, “DCE Daemon and Endpoint Map Service” on page 188 explains how the client RPC runtime gets an endpoint from the remote DCE daemon to form a fully bound handle used to contact the server application directly.

However, the server might support multiple object UUIDs, interfaces and protocol sequences. And multiple server machines may support the same application servers. So, when the client looks for binding handles, it might obtain handles to several compatible servers. It depends on the ***binding method*** (see 10.3.3, “Binding Methods” on page 187) whether the client uses a random selection or whether it wants to control, through several management calls to CDS and the remote endpoint mapper, what specific binding handle it wants to use.

When making a remote procedure call, the client runtime provides information about the client to the server runtime. This information, known as ***client binding information***, includes the address where the call originated, the RPC protocol used by the client, the object UUID that a client requests, and authentication information. A server application can use the client binding handle to ask the RPC runtime about this information.

The binding handles are annotated with security information. The server adds the levels of security its supports to the handles registered with its RPC runtime. The client adds the requested security level and its own identity into the binding handle used to contact the server. This is explained in 10.4, “RPC and Security” on page 191.

10.3.2 Name Service Interface

The Name Service Interface defines several kinds of Cell Directory Service (CDS) entries that can be made in the namespace. The NSI interface provides APIs which allow servers to export binding information into CDS objects and clients to import them.

Binding information in CDS does not have endpoints; binding handles returned by NSI calls are *partly bound*. Dynamic endpoints can be different every time they are assigned; so information in CDS would have to be updated very frequently if CDS were to store *fully bound handles* containing endpoints. Only well-known endpoints are stored in CDS. In this case, clients obtain *fully bound handles*.

10.3.2.1 RPC Entries in CDS Objects

Application servers can store binding information in CDS leaf objects. Several servers may offer the same services and are interchangeable. Multiple server entries may be grouped in specific CDS objects to provide for a randomized or prioritized list of compatible servers. This allows for server replication and load balancing.

RPC can store the following entities in a CDS leaf object:

- An ***RPC server entry*** stores binding information and object UUIDs for an RPC server.
- An ***RPC group*** stores names of one or more RPC server entries or RPC groups, which should represent compatible or interchangeable server instances. By looking up binding information in a group rather than in a specific server entry, a client can initialize a random search for a compatible server. This allows an administrator to provide for load balancing.
- An ***RPC profile*** is similar to the group. It contains other server entries, groups and profiles which allow a search for a compatible server binding. Other than group elements which are randomly searched, profile elements have an assigned priority which specifies a search path.

10.3.2.2 Searching The Namespace

NSI provides two methods for finding a server, the `rpc_ns_binding_import_*()` routines and the `rpc_ns_binding_lookup_*()` routines. Both operations search server entries for a compatible server.

The difference between *import* and *lookup* operations is that the lookup operations return a list of binding handles in the sequence in which they are stored in CDS, while the former returns just one randomly chosen binding handle at a time.

To search for a compatible server, the client must perform the following steps:

1. To start a search, a client calls the `rpc_ns_binding_*_begin()` routines, where the asterisk (*) stands for *import* or *lookup*. Input parameters for these calls are a CDS object name where the search begins along with the interface identifier and object UUID to specify the compatibility criteria for the requested server. The CDS entry can be any of the entries mentioned above in 10.3.2.1, “RPC Entries in CDS Objects.” This step returns a *name service handle* needed for the actual lookup operations.

2. The application calls the rpc_ns_binding_*_next() routines with the context handle created in the previous step. Each *next* operation returns another value (for the import operation) or list of values (a binding vector for the lookup operation).

3. The client may examine the binding handle or simply accept it when it comes in as a single handle. When a binding vector is returned, the client can examine the handles contained in the vector or call rpc_ns_binding_select() to randomly select one. If the client wants to examine a handle, it has to convert it to a string binding with rpc_binding_to_string_binding() because the binding handle is a pointer to an opaque data structure. In the string binding, the client could check, for example, the network address or the object UUID.

4. When the client does not want to accept or try a binding handle, it can go back to step 2.

5. When the client accepts a binding, it should delete the name service handle by calling the corresponding NSI rpc_ns_binding_*_done() operation.

10.3.3 Binding Methods

The client process needs a binding handle to execute an RPC call. In the simplest case, a programmer need not worry about binding handles. If they do not care to which (compatible) server instance an RPC call will be routed, they can use *automatic binding*. In this case, the client stub looks up binding information in CDS and creates the handle.

If the programmer wants full flexibility, he might want to select a server according to some application-specific criteria. The programmer is responsible to obtain a binding handle either via CDS lookups or by assembling a string-format binding and converting it to a binding handle. If they use the same handle for all RPC calls, they can use *implicit binding*. If they want to use different servers from the same client, they need to use the most complex method, that is *explicit binding*.

10.3.3.1 Automatic Binding

This is the simplest method of managing the binding for remote procedure calls since the client stub automatically manages the binding for the application code. The automatic method completely hides binding management from client application code. The client stub handles all needed operations to obtain a binding handle. If the client makes a series of remote procedure calls, the stub passes the same binding handle to these calls.

With the automatic method, a disrupted call can sometimes be automatically rebound. The client must specify the CDS object name for the CDS search in the ***RPC_DEFAULT_ENTRY*** environment variable. The environment variable can be set externally or within the program through a setenv() call.

10.3.3.2 Implicit Binding

This is a relatively simple method to manage a binding. With the implicit method, the client is responsible for obtaining a server binding handle and assigning it to a global variable specified in the interface definition. When calling a remote procedure call, the client stub passes this global binding handle to the runtime.

10.3.3.3 Explicit Binding

This is a more complex and flexible method of managing a binding. As with the implicit method, the client application code creates a binding handle. In the explicit method, however, this binding handle is supplied by the application code as a parameter to each remote procedure call.

By allowing a client to manage bindings for individual calls, the explicit method enables clients to meet specialized binding requirements. A client can be multithreaded and use several different remote services at the same time.

10.3.4 DCE Daemon and Endpoint Map Service

A DCE server host may run several RPC server applications. For an RPC client to connect to a particular RPC server, it needs to know the:

- Network address of the server machine
- Address of the process serving the call, called an ***endpoint***

An endpoint is a transport-layer address to the application server. The endpoint address is specific to the transport protocol the application server will use. For example, in TCP/IP, the machine address is the IP address, and the endpoint is the port address. Together, they build an IP socket to which the server process is listening. Applications can listen to one or multiple endpoints. That is completely up to the application and usually depends on how many interfaces, protocols, object UUIDs, and manager routines it supports. A simple application server will support one endpoint.

An application can choose to use a *well-known* endpoint and listen to it without using the DCE daemon. However, this would require a lot of coordination effort between the different developers, vendors and manufacturers. For TCP/IP, port addresses are administered and assigned by the ARPANET Network Information Center. Therefore, applications better use *dynamic* endpoints. They are assigned when a server application starts. The ***endpoint map service*** manages the current list of endpoint addresses. As part of the RPC binding process between a client and a server, the endpoint mapper tells the client which port it should use to connect to the desired server process.

The endpoint mapper, shown in Figure 66 on page 181, is a service of the DCE daemon (dced) in OSF DCE 1.1. It used to be part of the RPC daemon (rpcd) in OSF DCE 1.0.x, which is no longer available.

The dced process always uses the same network endpoint; so its process address is well known. It listens on one endpoint for each protocol. For example, if a host supports TCP/IP and UDP/IP, dced will listen on one TCP and one UDP socket (port 135 on both) for client requests.

In addition to the endpoint mapper service, the dced provides the DCE host services that are able to control and manage other DCE servers, including application servers. The services include tasks such as security validation, starting and stopping individual servers, monitoring a running server′s states, and managing server passwords remotely. This is a new function in OSF DCE 1.1.

10.3.5 Entry-Point Vector

When a call arrives at the server, the server must be able to determine which routine to call. The manager is part of the server implementation that contains all procedures defined in the interface.

For each interface supported by the server, there is an *entry-point vector* (EPV) that contains a list of addresses of the remote procedures provided by the manager. It is an array of function pointers. A manager EPV must contain exactly one entry point for each procedure defined in the interface definition. A default manager EPV is typically generated into the stub code by the IDL compiler. A server that does not handle several managers can use the default EPV provided in the stub.

For any additional manager of the same RPC interface, the server must create and register a unique manager EPV. Each manager must also be associated with a distinct type object. See 10.3.6, “Object UUIDs and Manager Types” for more details on multiple manager types for the same interface.

For example, the bank demo that comes with the DCE application development tools defines several interfaces, and for each interface, it defines an entry point vector (a vector of pointers to procedures):

```
globaldef admin_v1_0_epv_t admin_epv = { admin_open_bank,
                     admin_write_bank,
                     admin_create_acct,
                     admin_delete_acct,
                     admin_inquire_acct
                       };

globaldef trans_v0_0_epv_t res_trans_epv = { res_deposit,
                             res_withdraw
                       };
```

When it calls the rpc_server_register_if() function to register an interface, it also passes the respective entry point vector as an argument.

10.3.6 Object UUIDs and Manager Types

An RPC object is an entity that an RPC server defines and identifies to its clients. Frequently, an RPC object is a distinct computing resource, such as a particular database, directory, device, or processor. Applications can use RPC objects to differentiate between RPC interfaces that operate on specific resources. An RPC object can also be an abstraction that is meaningful to an application, such as a service or the location of a server.

RPC applications generally use RPC objects to enable clients to find and access a specific server. When servers are completely interchangeable, using RPC objects may be unnecessary. However, when clients need to distinguish between two servers that offer the same RPC interface, RPC objects are essential.

As we mentioned before, a server can offer multiple implementations (or managers) of the same RPC interface. It can overlay the set of procedures pointed to by an EPV with another set of procedures. It needs to define another manager EPV that points to different functions. A manager is associated with a type UUID and must also be associated with an object UUID which can be referenced by client. During server initialization, the rpc_object_set_type()

routine has to be called for every manager type-to-object UUID association. For every manager, the `rpc_server_register_if()` has to be called which registers a particular manager EPV with the interface.

By choosing another object UUID, the client can choose another set of procedures executed on behalf of its calls. For those readers who are familiar with object oriented programming, the interface can be compared with a parent class containing only virtual methods, and manager types can be compared with classes that inherit from the parent and have to implement the methods. The term *object UUID* can be seen as a pointer to a remote object with its methods.

10.3.7 Putting It Together: A Summary of RPC Call Routing

We discussed (partially and fully bound) binding handles, endpoints, binding methods, NSI lookups, object UUIDs, and RPC managers. Here, we will summarize how a client finds a particular routine in a remote application server.

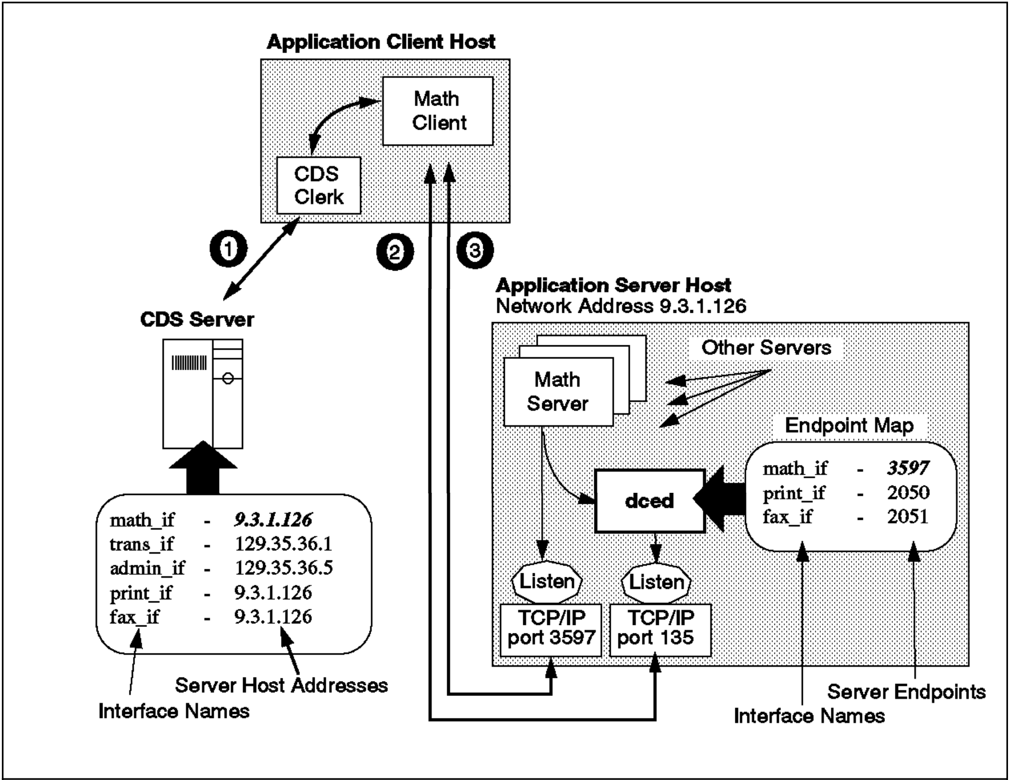


Figure 68. Steps Involved in Finding a Server

Figure 68 illustrates a simplified process of a client searching for a server. It performs the following steps:

1. Looking up a binding in CDS

   The client sends a request to its local CDS client (cdsclerk) to look up an entry in the name space. If it does not have the information in its own cache, the local cdsclerk contacts the CDS server to search for compatible binding information. Depending on the calls it used, the client gets one randomly selected binding handle at a time or a binding vector. The handles are partly bound, meaning they do not have an endpoint.

If the RPC entry used to start the search in CDS was a group or profile, the look up can go over multiple server entries providing a random selection of one of multiple machines running the same application servers.

2. Contacting the remote endpoint mapper

   The client selects a binding handle and issues the RPC call. Since the handle is a partly bound handle, the call goes to the remote DCE daemon listening on the well-known endpoint 135. The endpoint mapper function of the DCE daemon looks up the endpoint registered for the requested interface, object and protocol. It adds the endpoint to the binding handle and returns it to the client′s RPC runtime.

3. Executing the RPC

   With the fully bound handle, the client′s RPC runtime then directly calls the server process listening to the endpoint. The server′s RPC runtime takes the call, checks what object and interface is being addressed, chooses the right manager EPV, and dispatches the call to the procedure point to by the EPV.

If the client uses automatic binding, it just calls the desired function, and the RPC runtime performs all the above look-up and selection steps.

10.4 RPC and Security

DCE RPC supports authenticated communications between clients and servers. Authenticated RPC is provided by the RPC runtime facility and works with the authentication and authorization services provided by the DCE security service.

Before authenticated RPC can be used, the application server registers its principal name and the supported authentication service with its RPC runtime. A server usually assumes its own (authenticated) login identity during its initialization. It performs the equivalent of a user login by specifying its DCE account name and password stored in the local keytab file. See also 10.5.6, “Developing a Basic Server” on page 199 for details on the server initialization steps.

A client usually runs with the login context of the user that called it. To use authenticated RPC, a client must specify the server principal name and establish the authentication service, protection level and authorization service that it wants to use in its communications with a server. The client does this with a call to `rpc_binding_set_auth_info()`, which adds this security information to the server binding handle. The client then uses this extended binding handle in its further RPC calls.

The client can call `rpc_mgmt_inq_server_princ_name()` with the server binding handle established so far if it does not know the server principal name. This procedure contacts the server′s RPC runtime to query the registered server principal name.

10.8, “Putting It All Together: Initialization, Routing and Execution” on page 207 explains at what stage of the client/server communication security routines are called. The rest of this section describes the authentication service, protection level and authorization service. At the end of this section, there is a caveat on server key management and secret key authentication.

10.4.1 Authentication Service

The DCE Remote Procedure Call (RPC) programming facility is connected with the security components to provide mutual client/server authentication of principal identity.

The RPC mechanism includes automatic use of the DCE Kerberos authentication protocol. DCE RPC offers several types of authentication protocols.

When a client establishes authenticated RPC, it must indicate the authentication service that it wants to use. The server must have registered the same service with a call to `rpc_server_register_auth_info()`. A server can register more than one authentication service.

The possible values for the authentication service are the following:

- None. No authentication.
- DCE_secret. DCE shared-secret key authentication.
- DCE_public. DCE public key authentication.
- Default. DCE default authentication service (which is DCE_secret).

10.4.2 Level of Protection

When a client establishes authenticated RPC, it can specify the level of protection to be applied to its communication with the server. The protection level determines the degree to which client/server messages are actually encrypted. As a rule, the more restrictive the protection level, the greater the impact on performance.

The following protection levels are available:

- None. No communication protection.
- Connection. Performs an encrypted handshake the first time the client communicates with the server.
- Call. Attaches an encrypted verifier only at the beginning of each remote procedure call over connectionless communication. This level does not apply for TCP connections.
- Packet. Attaches a verifier to each message sent over the network to make sure all messages are from the expected client.
- Packet Integrity. Ensures and verifies that no messages have been modified by computing and encrypting a checksum over each message.
- CDMF Privacy. Encrypts RPC arguments and data in each call using CDMF.
- Packet Privacy. Encrypts RPC arguments and data in each call using DES.

Encryption is done with the session key, which is only known by the client and the server for which the service ticket was issued. More explanations on session keys and how they are obtained and managed are in 10.4.4, “Key Management and Secret Key Authentication” on page 193 below.

On most platforms, encryption is done with the data encryption standard (DES) algorithm which cannot be exported outside the U.S. in a user-accessible form. This means that DES can be used for protection levels up to and including *Packet Integrity*, but not for *Data Privacy*.

On the AIX platform, there is a User Data Masking Facility, which is still referred to as Common Data Masking Facility or CDMF. CDMF allows you to encrypt user data in RPCs using DES with a 40-bit key instead of the standard 52-bit key. Since this makes the encryption weaker, it has less export restrictions from the USA. It is a good solution for non-U.S. customers who want increased privacy, but cannot have an export license for full DES.

It is possible that data privacy will not be usable due to laws in some countries (France, UK) which restrict encrypted data from crossing their borders; in this case, only data integrity can be used.

10.4.3 Authorization

Authorization is the mechanism that allows the server to control client access to a resource. The authorization process is application dependent. It is up to the server side of the application to implement the appropriate authorization checking it needs. The authorization process involves the matching of clients′ privilege attributes against the permissions associated with an object.

The server can ask the RPC runtime for the privileges associated with a client. Authenticated RPC supports the following operations for making client authorization information available to servers for access checking:

- None. No authorization information is provided to the server.
- Name. Only the client principal name is provided to the server. This type of authorization is called *name-based* authorization.
- DCE. The client′s DCE Extended Privilege Attribute Certificate (EPAC) is provided to the server with each remote procedure call made using the binding parameter. The EPAC contains the principal name and a list of groups of which the principal is member. The EPAC also contains the name and group memberships of a principal in the delegation chain and any extended attributes that apply to the principal.

Once the server obtains the client′s authorization information, is up to it what to do with this information. To perform an authorization check, it can implement its own authorization scheme based on information stored on a file or a database; or it can even compare the information received with some hard-coded authorization information. The server can also implement an ACL manager that allows a security administrator to maintain permission in a standardized way outside of the program by using the acl_edit program or dcecp. The server calls the appropriate ACL manager for the type of object requested by the client and passes the manager the client EPAC. The manager compares the client authorization data to the permission associated with the object and either refuses or permits the requested operation. If the EPAC contains a delegation chain, the ACL manager grants access for the requested operation only if all principals in the delegation chain have the necessary permissions on the object that is the eventual target of the operation.

10.4.4 Key Management and Secret Key Authentication

When installing an application server that needs to run authenticated RPCs, it has to register a server principal name with the RPC runtime. This is done through the rpc_server_register_auth_info() call. The client specifies the server principal it wants to connect to with the rpc_binding_set_auth_info() call. The client and server runtimes then perform the mutual authentication.

Authentication is basically done with the ***server key***. The server key is the server′s encrypted password. How does the server supply its password? The administrator has to create a principal and an account in the Security Registry. The administrator then uses the rgy_edit′s `ktadd` command or the dcecp′s `keytab add` command to add a password, which can be randomly generated. The password is encrypted by the local Security Service runtime and saved in the local ***keytab file*** as well as in the Security Registry. This is the server key.

The client RPC runtime requests a ***service ticket*** from the Security Service. This ticket contains the client′s extended privilege attribute certificate (EPAC) and a ***session key*** for the upcoming client/server communication. The EPAC contains the principal name and groups of which this principal is a member. This ticket is encrypted with the server′s session key. So, the client cannot decrypt and change it to its own liking. Together with the (unreadable) ticket, the client also is sent the session key. Of course, this communication between client and Security Service is itself encrypted.

The client RPC runtime encrypts the RPC call with the session key and sends it to the server′s runtime together with the ticket. The server immediately challenges the client by sending it a randomly generated number which the client has to encrypt with the session key and return to the server.

The server′s runtime obtains the server′s key from the local keytab file and decrypts the ticket, thereby learning the session key and the client′s EPAC. The random number is decrypted, and if it matches, everything is set for authenticated RPC. The session key is used in further communication over this binding. In theory, the server can register more than one principal and authentication service. Then, each of these principals has to have a password (key) in the keytab file.

For ***automated key management***, servers spawn off a thread and call the `sec_key_mgmt_manage_key()` routine, which checks the expiration date of a particular principal′s key and replaces it in time with another randomly generated password. This has nothing to do with ***ticket lifetime***. If the ticket expires, the server throws the session key away, and the client RPC runtime has to obtain a new ticket.

The server can use (one of) its registered principal name(s) and the accompanying key to perform the equivalent of a user login and create its own login context. Long-running servers (and clients) should also spawn off a thread to monitor the ticket lifetimes and refresh tickets, if necessary.

See also 3.2.3.6, “Keys and Key Management” on page 57 for more information.

10.5 Developing an RPC application

This section explains the elements involved in the RPC application development process.

10.5.1 Universal Unique Identifiers

UUIDs are guaranteed to be unique numbers. They have to be unique for all time (at least for the life time of the companies using them). A UUID is a 16-byte structure composed of the current time of day, measured in hundreds of nanoseconds since January 1,1970, and the IEEE802 network hardware address.

A well-behaved clock will never move backwards. This is very important for guaranteeing the uniqueness of UUIDs. The UUID generator keeps track of the last UUID generated. If it detects that the clock has moved backward, it adjusts for this with the *clock sequence number* field which is modified each time the clock is found to have moved backwards.

There are three variants of the UUID specification. The variant in use is identified in the *reserved* field (this allows the variants to coexist). The version number field is the version of the variant.

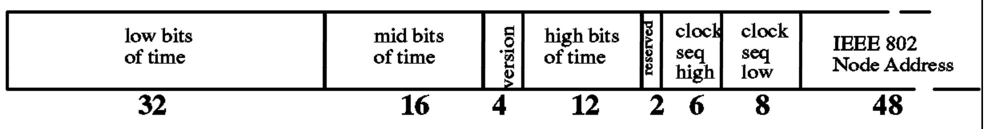


Figure 69. UUID Structure

The applications use UUIDs to identify several kinds of entities, such as:

1. Interface UUIDs as a required part of the RPC interface definition
2. Object UUIDs to identify resources a server manages
3. Type UUIDs as classes of RPC objects with their own manager routines

DCE provides a utility to create UUIDs. In addition to plain UUIDs, the UUID generator (`uuidgen` command) can also create an RPC interface definition template file (an `.idl` file) containing the newly generated interface UUID. To create template for an IDL file and display it, run the following commands:

```
# uuidgen -i -o Bank.idl
# cat Bank.idl
[
    uuid(9967703a-b821-11ce-9060-10005a4f4629),
    version(1.0)
]
interface INTERFACENAME
{
}
```

10.5.2 Interface Definition File

The first step in developing an application is to write an interface definition file. It declares the remote procedures in its own interface definition language (IDL), which is similar to the C programming language with the addition of attributes. The IDL file is pre-compiled with an IDL compiler to create the client and server stub code. This is explained in 10.5.4, “IDL Compiler” on page 197.

In 10.5.1, “Universal Unique Identifiers,” we describe how to create a skeletal interface definition file with the `uuidgen` command. This file needs to be edited to contain all the data types, attributes and procedure call declarations that make up the interface. The RPC interface definition contains two basic components:

- An interface header that contains a UUID, interface version numbers and an interface name.
- An interface body that declares any application specific data types and constants as well as operation declarations with their parameters and return values.

In local programs, the operations, the parameters and the structures are stored in the same process′s memory and are easily accessible via pointers. In distributed applications, you need to worry much more about the efficiency of data access or transfer between a calling and a called function. IDL uses attributes to define the behavior of a particular client/server application. For instance, attributes specify how arrays and pointers are to be treated and how the portion of memory referenced by them is transmitted. The pointer_default attribute in the example below defines the type of pointer:

```
[uuid(88fdbace-f5a3-11c9-9999-02608c2ea88e), pointer_default(ptr),
 version(0)]
interface trans
{
        import ″acct_type.idl″;

        error_status_t deposit(
                [in] handle_t h,
                [in] char name[30],
                [in] long amount
        );

        error_status_t withdraw(
                [in] handle_t h,
                [in] char name[30],
                [in] long amount
        );
}
```

For every parameter in the list, the programmer also has to specify whether it is for input, output or both. The example above has only input parameters (*in*). The return value of the functions tells the caller whether a deposit or withdrawal to/from the account specified by the input parameters was performed or failed.

The interface header is all made up of attributes. In addition to the UUID, version, interface name, and pointer_default you can also define a well-known endpoint.

IDL defines many new data types, such as binding handles and pipes. Pipes are used to transfer large amounts of data. Another feature of IDL are context handles. They are used to maintain the state or value of variables on the server between successive calls of the same client.

10.5.3 Attribute Configuration File (ACF)

We can tailor how the IDL compiler creates stubs. This can be done differently for clients and servers without changing the interface definition. The IDL file defines the interface, whereas the ACF can, for instance, instruct the IDL compiler to include only a subset of functions into a specific client stub or how it should handle marshalling and unmarshalling.

The ACF file can also define a specific binding method for a server. Defining the explicit binding method in the ACF, for instance, instructs the IDL compiler to add a binding handle to each procedure without having to specify it in the IDL file. In this way, another client could be built for the same interface using automatic binding. The comm_status and fault_status attributes are examples of ACF implicitly adding parameters to the parameter list.

We associate the filename.idl file with *filename.acf*.

10.5.4 IDL Compiler

The DCE IDL compiler processes the RPC interface definition written in the *.idl* and *.acf* file and generates header files and stub object code. You can also request the IDL compiler to keep the intermediary stub source code written in ANSI C.

The IDL compiler also generates a data structure called ***interface specification*** which contains identifying and descriptive information about the compiled interface. It creates a companion global variable, the ***interface handle***, which is a reference to the interface specification.

The interface handle is directly used by runtime operations to obtain required information about the interface, such as UUID and version number. Servers use the interface handle during their initialization to register the interface, and clients use it to locate a compatible server.

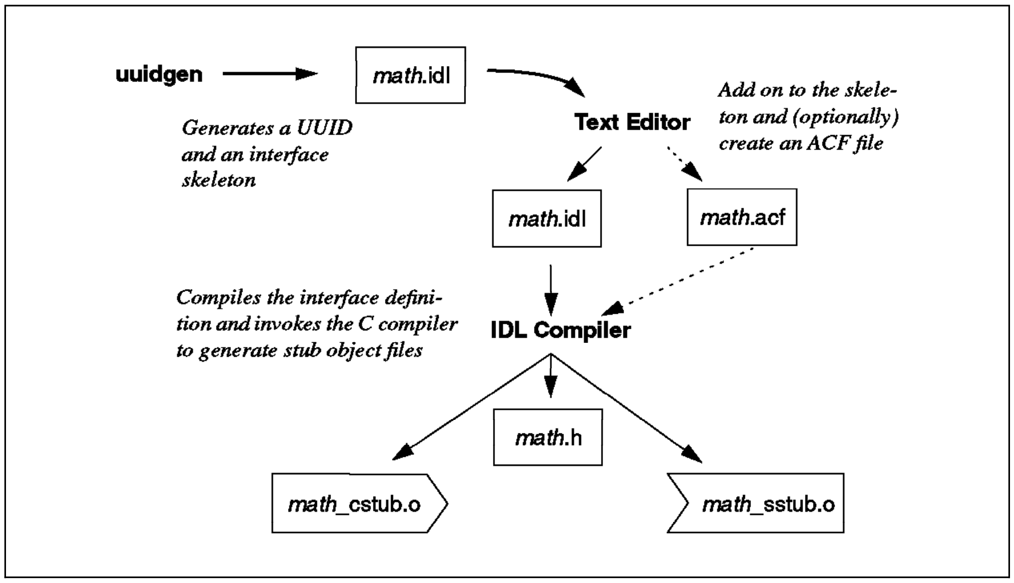

uuidgen
math.idl
Text Editor
Add on to the skeleton and (optionally) create an ACF file
Generates a UUID and an interface skeleton
math.idl
math.acf
Compiles the interface definition and invokes the C compiler to generate stub object files
IDL Compiler
math.h
math_cstub.o
math_sstub.o

Figure 70. IDL Compiling

Figure 70 shows the steps needed to compile a RPC interface. After creating the *.idl* file (and optionally the *.acf* file), you can run the IDL compiler to generate the client and server stub code. For example, to generate the stub code for the math interface, you run the following command:

```
# idl math.idl -keep c_source
```

Besides the regular header (math.h) and stub files (math_cstub.o and math_sstub.o), the compiler is instructed to also keep the ANSI C source files for the stubs (math_cstub.c and math_sstub.c).

10.5.5 Developing a Basic Client

The complexity of a client program depends on the binding method. When using automatic binding, the simplest client does not have to worry about finding and contacting a server. It just uses the remote call; the stub and runtime do the rest.

For example, when a client wants to use the product() call as defined in the *math* interface shown in 10.1, “What is RPC?” on page 173, it can do so using the following code:

```
#include ″math.h″

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

 if (ac != 3)  {
    printf(″Usage: %s number_1 number_2\n″,argv[0]);
    exit (1);
 }
 printf(″ %s + %s = %d\n″,argv[1],argv[2],product(atoi(argv[1]),atoi(argv[2])));
}
```

The program has to be linked with the math_cstub.o. At runtime, the RPC_DEFAULT_ENTRY environment variable must be set to a directory service entry that contains binding information about the math server. If the client were to use implicit or explicit binding, it would have to use the NSI search functions described in 10.3.2.2, “Searching The Namespace” on page 186.

If the client wants authenticated RPC, it needs to annotate the binding handle with the name of the server principal and the requested security levels. 10.4, “RPC and Security” on page 191 discusses how to achieve this.

Figure 71 on page 199 shows the steps needed to develop a DCE RPC client.

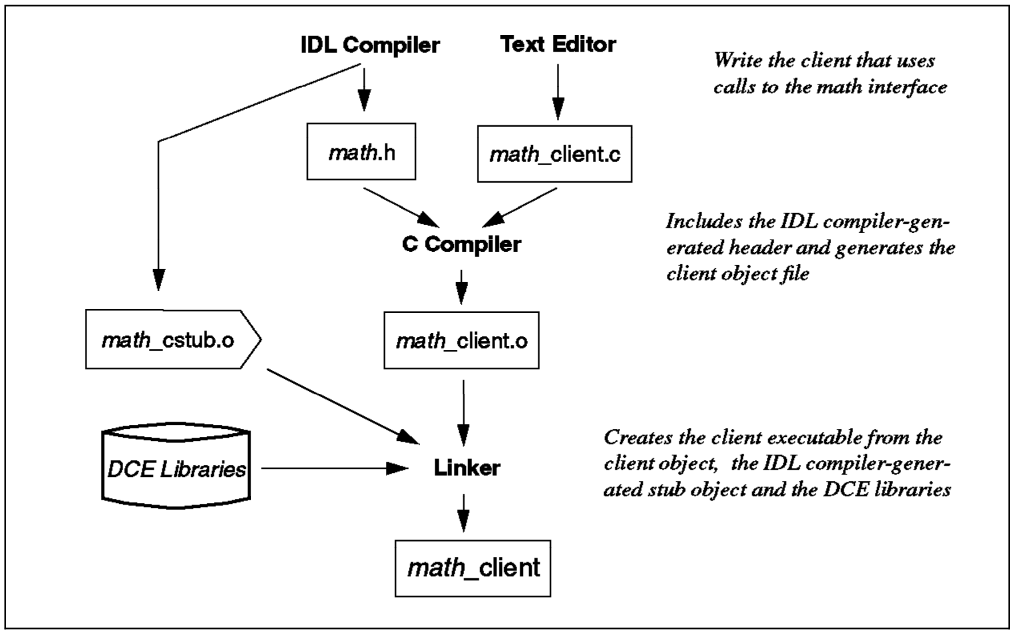

IDL Compiler
Text Editor
Write the client that uses calls to the math interface
math.h
math_client.c
C Compiler
Includes the IDL compiler-generated header and generates the client object file
math_cstub.o
math_client.o
DCE Libraries
Linker
Creates the client executable from the client object, the IDL compiler-generated stub object and the DCE libraries
math_client

Figure 71. Client Development Tasks

10.5.6 Developing a Basic Server

The server implements the RPC procedures defined in the RPC interface definition file. As introduced in 10.2.5, "RPC Application Server and Manager" on page 180, a server consist of two parts:

1. A server part that initializes the server and defines its capacities
2. A manager part that implements the procedures as defined in the IDL

The development steps illustrated in Figure 71 are the same for servers; instead of client stub and code, you would put server stub and server/manager code. So, we want to concentrate on the server initialization in this section. Basically, this involves the following tasks in the listed sequence:

1. Creating a login context: several `sec_login_*()` commands

   A server basically runs under the identity of the user who started the process. If the root user started the process without having done a DCE login, the server runs under the authenticated machine principal. This might be sufficient. However, if the server wants to establish its own login identity (login context), it performs the equivalent of a DCE user login with its own user account. The password (actually the key, which is the locally encrypted password) is taken from the local keytab file.

   This step is optional, but be aware that the principal name in the login context is used when this server turns into a client and wants, for instance, access any object protected by ACLs.

2. Assigning manager types to objects: `rpc_object_set_type()`

   The association of a *manager type* to a manager is a mechanism for overlaying one manager offering a set of procedures with another manager offering the same (but differently implemented) procedures. An object UUID has to be assigned to every manger type that serves client requests for the same interface. Clients can specify an object UUID to select a specific manager. We explain objects in more detailed in 10.3.6, “Object UUIDs and Manager Types” on page 189.

3. Registering interfaces: `rpc_server_register_if()`

   Registering an interface informs the runtime that the server is offering that interface. The first parameter is the interface handle as found in the IDL compiler-generated header file. Further parameters are a manager type UUID and the manager EPV that is pointing to the functions implemented by this manager. Both can be NULL. If the server offers more than one interface and/or more than one manager for the same interface, it must call this routine several times.

4. Specifying supported protocols: `rpc_server_use_all_protseqs()`

   The server can inquire about the protocols sequences that the local RPC runtime supports before it calls this routine. You must use at least one protocol sequence to receive remote procedure calls. After a server has passed the supported protocols sequences to the RPC runtime, the RPC runtime creates the necessary endpoints and server binding handles, one for each protocol sequence, regardless of how many interfaces or managers you specified in the step before. At this point, a binding handle consists of a protocol sequence, a network address and an endpoint. All interfaces of the same server are served by the same binding handle(s).

   A server can supply a well-known endpoint. When using dynamic endpoints in TCP/IP, this step creates new sockets which will be served by this application server. The number of sockets created also depends on the size of the requested *request buffer*, since a socket can only queue five requests.

5. Obtaining a list of binding information: `rpc_inq_server_bindings()`

   The server can query the RPC runtime about the binding handles created on the previous step. Each binding handle represents one way to establish a binding with the server. The list of binding handles is needed as input to the next steps which advertise bindings.

6. Registering endpoints with dced: `rpc_ep_register()`

   A server can use a well-known or dynamic endpoints with any protocol sequence. To use the dynamic endpoints, a server must register the server′s binding handles with the local endpoint mapper (managed by the dced daemon). With this call, the server has to specify one interface handle, the binding handles obtained in the step before, all object UUIDs that belong to this interface, and a textual annotation. For every combination of binding handle and object UUID, the DCE daemon creates an entry in its endpoint map for the particular interface handle that goes with this call. If the server registers more than one interface, it has to run this call once for each interface.

   A server does not need to register well-known end points; however, by registering well-known endpoints, the server ensures that clients can always obtain them even if they do not know them.

7. Advertising the server in CDS: `rpc_ns_binding_export()`

   This step exports the binding information to one or more entries in the CDS namespace.

8. Registering authentication information: `rpc_server_register_auth_info()`

Registers the supported authentication level with the RPC runtime together with the server principal name and information on how the runtime can access the server′s key to perform the secret-key authentication.

The server can register multiple authentication levels and principal names by calling the routine more than once. The client can request any of these registered server principal names and establish an authenticated communication. In fact, the server principal(s) registered here can be different from the principal name of the process′s login context.

9. Listening for remote procedure calls: `rpc_server_listen()`

   When the server is ready to accept remote procedure calls, it starts listening. With this step, it specifies the maximum number of calls it can concurrently execute.

10.5.7 Servers with Multiple Interfaces

As we said before, a server can register multiple interfaces. For example, the bank demo uses two interfaces, an *admin* and a *trans* interface. These interfaces act on four objects: a checking account, a savings account, a CD account and an IRA account. The checking and savings account have a type of *unrestricted*. The CD and IRA account have a type of *restricted*. Each of these types have different manager code to implement the *deposit* and *withdraw* bank transactions. This eventually results in restricted withdrawal and deposit operations, and unrestricted withdrawal and deposit operations, which are selected depending on the type of the object (account type) they are applied to.

The server code has three IDL files. The acct_type.idl file is just a definition of files.

To create the server, you have to run the IDL compiler over these `.idl` files:

```
# idl acct_type.idl -I/usr/include/dce
# idl admin.idl -I/usr/include/dce
# idl trans.idl -I/usr/include/dce
```

This generates the header files and the compiled stub code. The server code must include both header files (admin.h and trans.h). The server must register both interfaces with the RPC runtime. Since the trans interface implements two managers, it registers this interface twice with different type managers (restricted and unrestricted type) and manager EPVs:

```
    rpc_server_register_if(admin_v1_0_s_ifspec, (uuid_p_t)NULL,
                  (rpc_mgr_epv_t)&admin_epv, &st);

    rpc_server_register_if(trans_v0_0_s_ifspec, &rest_type,
                  (rpc_mgr_epv_t)&res_trans_epv, &st);

    rpc_server_register_if(trans_v0_0_s_ifspec, &unrest_type,
                  (rpc_mgr_epv_t)&unres_trans_epv, &st);
```

When compiling the server code, it must be linked with the admin_sstub.o file and the trans_sstub.o file.

10.5.8 Using Manager Types

Manager types allow you to have different implementations of the same interface. The bank demo we explained in the previous section uses two manager types for the transaction RPC interface. It defines restricted and unrestricted transactions. Depending on the object that the client is requesting, the RPC runtime will select the appropriate manager (EPV) to handle the RPC. The server code defines a specific UUID for each object (accounts) and for each type. To be able to use manager types, it has to associate an object UUID to each manager type:

```
rpc_object_set_type(checking_obj, unrest_type, &st);
rpc_object_set_type(savings_obj, unrest_type, &st);
rpc_object_set_type(cd_obj, rest_type, &st);
rpc_object_set_type(ira_obj, rest_type, &st);
```

This has to be done at first, before registering the interface. Then it registers the interfaces, as shown in the previous section 10.5.7, "Servers with Multiple Interfaces" on page 201.

The client side will set an object UUID according to the account it will operate on. The object UUID is added to the binding handle before the RPC is made:

```
rpc_binding_set_object(trans_bindinghandle[bank],&acct_uuid,&st);
   .
   .
st = deposit(trans_bindinghandle[bank],name,amt);
```

When the server RPC runtime receives this RPC, it will route it to the appropriate manager, depending on the object UUID.

10.6 RPC Administration

Only few administrative tasks must be performed when running a distributed application using RPC. The application server executes most of these tasks when it first starts. Non-automated RPC administration is minimal. It is essential that each DCE machine has a DCE host daemon running on it. The DCE daemon is the first process started when DCE is brought up. When managing RPC applications, an administrator might have to do the following:

- Create security registry information for the server (principal, account)
- Create a keytab entry with password for the server
- Create namespace entries for the server′s bindings
- Change the ACLs of objects the server principal has to access
- Manage the endpoint mapper of the server′s DCE daemon

The administrator might have to perform some of these tasks with DCE commands. More likely, an application will provide installation and configuration tools that do the job.

The first step after installing an RPC application is to create a principal and an account in the Security Registry. This must be the same name as the server exports into its RPC runtime upon initialization. The administrator then creates a keytab entry so that the server can automatically authenticate itself with the DCE Security Service. See 10.4, "RPC and Security" on page 191 for more information.

The rest of this section describes how to manage the namespace and the endpoint map. If the server has to access any objects that are protected by ACLs, you also need to add an entry for the server principal. One example of this is the CDS directory to which the server wants to export its bindings.

10.6.1 Managing CDS Entries for RPC

An administrator may be involved in registering servers in the namespace, but this can also be done by the server itself upon initialization. Otherwise, the administrator might have to use the dcecp command (OSF DCE 1.1) or the rpccp command (DCE 1.0.x), which is still available, to manually register this information. An application can provide a configuration tool (or a script written in dcecp) to create static entries in the namespace.

Authorized individuals can add entries to and remove them from the namespace, or they can add information to and remove it from those entries. In the example below, we assume that there are two CDS directories: /.:/home and /.:/servers. The following steps create an RPC server entry and an RPC profile containing this server entry:

1. Login as cell_admin and start a dcecp shell:

   ```
   # dcelogin cell_admin <password>
   # dcecp
   dcecp>
   ```

2. Create two UUIDs and assign them to variables:

   ```
   dcecp> set printer_intf [list [uuid create] 1.1]
   b367ea90-9cf7-11ce-ba29-10005aa86e2d 1.1

   dcecp> set laser_printer [uuid create]
   96b316fe-9c66-11ce-ae77-10005aa86e2d
   ```

   The printer_intf variable defines the interface UUID with version 1.1 for the printer. The laser_printer variable specifies an object UUID for the laser printer object.

3. Create a new CDS object which can later be used as a server entry, a group entry or a profile entry:

   ```
   dcecp> rpcentry create /.:/servers/PrintServer
   ```

4. Convert the CDS object into an RPC server entry with the printer interface version 1.1, the UDP and TCP protocol sequences, the IP address of the server machine, and the object UUID for the laser printer:

   ```
   dcecp> rpcentry export /.:/servers/PrintServer -interface \
   $printer_intf -binding {ncacn_ip_tcp:9.3.1.126 ncadg_ip_udp:9.3.1.126} \
   -object $laser_printer
   ```

5. Check the entry:

   ```
   dcecp> rpcentry show /.:/servers/PrintServer
   {b367ea90-9cf7-11ce-ba29-10005aa86e2d 1.1
     {ncacn_ip_tcp 9.3.1.126}
     {ncadg_ip_udp 9.3.1.126}}
   {96b316fe-9c66-11ce-ae77-10005aa86e2d}
   ```

6. To add this server entry to an RPC profile, issue the following command:

   ```
   dcecp> rpcprofile add /.:/home/UserProfile -member /.:/servers/PrintServer \
   -interface  $printer_intf -priority 3 -annotation "1st floor laser printer"
   ```

7. An import operation, starting with the profile just created, will return the bindings for the printer interface. You can check this with the following command available on AIX:

```
rpcresolve -n /.:/home/UserProfile -p -s
(P) /.:/home/UserProfile
    (E) element     : /.../itsc7.austin.ibm.com/servers/PrintServer
        interface id: b367ea90-9cf7-11ce-ba29-10005aa86e2d,1,1
        priority    : 3
        annotation  : 1st floor laser printer
    (S) /.../itsc7.austin.ibm.com/servers/PrintServer
        (O) 96b316fe-9c66-11ce-ae77-10005aa86e2d
        (I) b367ea90-9cf7-11ce-ba29-10005aa86e2d,1.1
            (B) ncacn_ip_tcp:9.3.1.126[]
            (B) ncadg_ip_udp:9.3.1.126[]
```

Since the RPC entries are registered in the CDS namespace, we can also use normal CDS commands to display the entry. The output is less meaningful for RPC, though, but it shows UUIDs and timestamps. For example, enter the following cdscp and dcecp commands:

```
# cdscp show object /.:/servers/PrintServer

                  SHOW
                OBJECT   /.../cell1.itsc.austin.ibm.com/servers/PrintServer
                    AT   1995-07-06-14:32:06
       RPC_ClassVersion = 0100
        RPC_ObjectUUIDs = fe16b396669cce11ae7710005aa86e2d
                CDS_CTS = 1995-07-06-19:27:16.238528100/10-00-5a-4f-46-29
                CDS_UTS = 1995-07-06-19:27:26.009520100/10-00-5a-4f-46-29
              CDS_Class = RPC_Server
       CDS_ClassVersion = 1.0
             CDS_Towers = :
                  Tower = ncacn_ip_tcp:9.3.1.126[]
             CDS_Towers = :
                  Tower = ncadg_ip_udp:9.3.1.126[]
# dcecp
dcecp>  object show /.:/servers/PrintServer
{RPC_ClassVersion {01 00}}
{RPC_ObjectUUIDs
 {fe 16 b3 96 66 9c ce 11 ae 77 10 00 5a a8 6e 2d}}
{CDS_CTS 1995-07-06-19:27:16.238528100/10-00-5a-4f-46-29}
{CDS_UTS 1995-07-06-19:27:26.009520100/10-00-5a-4f-46-29}
{CDS_Class RPC_Server}
{CDS_ClassVersion 1.0}
{CDS_Towers
 {050013000d90ea67b3f79cce11ba2910005aa86e2d01000200010013000d045d888aeb1cc9119f
e808002b1048600200020000000001000b0200010001000702000000001000904000903017e}
 {050013000d90ea67b3f79cce11ba2910005aa86e2d01000200010013000d045d888aeb1cc9119f
e808002b104860020002000000001000a0200000001000802000000001000904000903017e}}
```

Most application servers also perform clean-up tasks when they terminate. This includes unexporting the binding information from CDS. However, if the server dies or it does not implement the clean-up tasks, the administrator might have to remove the entries manually to prevent clients from further trying to bind to that server. Since the CDS namespace is a distributed namespace and each client has a local cache (stored by the cdsclerk), the administrator might refresh the caches of the CDS clerk to avoid clients from getting the binding information of this server. To do so on a particular machine, stop the CDS clerk and the server, if there is one, and delete the following files:

```
# rm /opt/dcelocal/var/adm/directory/cds/cds_cache.*
# rm /opt/dcelocal/var/adm/directory/cds/cdsclerk_*
```

10.6.2 Managing the Endpoint Map

DCE provides several commands to manipulate the local endpoint map. To show the endpoint map on the local machine, you can use the following command:

```
dcecp> endpoint show
{{object 07dfb17a-b54e-11ce-aaae-10005a4f4629}
 {interface {e1af8308-5d1f-11c9-91a4-08002b14a0fa 3.0}}
 {binding {ncacn_ip_tcp 9.3.1.68 135}}
 {annotation {Endpoint Resolution}}}
 .
 .
{{object 019ee420-682d-11c9-a607-08002b0dea7a}
 {interface {019ee420-682d-11c9-a607-08002b0dea7a 1.0}}
 {binding {ncadg_ip_udp 9.3.1.68 1204}}
 {annotation {Time Service}}}
```

The same information can also be displayed with the `rpccp show mapping` command of OSF DCE 1.0.x.

If servers terminate, they should remove their own map elements from the endpoint map to prevent clients from receiving stale data. If a remote procedure call uses an endpoint from an outdated map element, the call fails to find a server. The endpoint map service routinely removes any map element containing an outdated endpoint; however, a lag time exists during which stale entries remain. An endpoint can be deleted manually with the `endpoint delete` command of dcecp:

```
dcecp> endpoint delete -interface {83eb9d64-b7e3-11ce-80be-10005a4f4629 1.0} \
> -binding {ncadg_ip_udp 9.3.1.68 5555}
```

You can also create an entry, for example, if you have deleted an interface by mistake:

```
dcecp> endpoint create -interface [uuid create],1.0 -binding \
> {ncadg_ip_udp 9.3.1.68 5555} -annotation Prueba -object [uuid create]
dcecp> endpoint show -object 83ec0f60-b7e3-11ce-80be-10005a4f4629
{{object 83ec0f60-b7e3-11ce-80be-10005a4f4629}
 {interface {83eb9d64-b7e3-11ce-80be-10005a4f4629 1.0}}
 {binding {ncadg_ip_udp 9.3.1.68 5555}}
 {annotation Prueba}}
```

10.7 Network Computing System, iFOR/LS and DCE

DCE RPCs are based on the RPCs of Apollo′s Network Computing System (NCS) 2.0. NCS uses location brokers to find RPC servers the same way DCE uses the CDS and its namespace. The local location broker (LLB), which runs as the llbd daemon, maintains a database of the objects and interfaces exported by servers running on the host. In addition, it acts as a forwarding agent for requests. An llbd daemon must be running on hosts that run RPC servers. However, it is recommended to run an llbd daemon on every host in the network or Internet.

A client agent is running on every system. It is a set of library calls linked into the client and server code. When a client needs to know the location of server, it can ask a global location broker and then directly connect with the service. If it knows a hostname, the client agent can call that host′s LLB forwarding agent. The llbd process is listening on port 135. It manages an endpoint map just like the DCE daemon does for DCE RPCs. The llbd sends the full address information back to the client, which then makes the call directly to the service.

The software license management system used by AIX Version 4 products is Information For Operation Retrieval/License System (iFOR/LS). Through the use of encrypted keys, iFOR/LS can monitor the type and number of licenses used by stand-alone machines or by machines within a network.

The iFOR/LS software uses NCS version 1.5. The user applications that make license requests are NCS clients. The license server (netlsd) is an NCS server. Therefore, before an application can get a license from an iFOR/LS server, it must first communicate with a global location broker to find out where the license servers are running. The global location broker receives the client request and replies with the information necessary for the client to establish communications with the server. The local location broker (llbd) provides some data and communications management during the transaction.

The iFor/LS documentation also suggests running an llbd on client systems if contacting a glbd takes too long. This is because some applications first try to look up a local LLB and try to get to the glbd only after a timeout.

The dced and the llbd daemons both listen on port 135; so, both servers cannot be started simultaneously. NCS applications can coexist with DCE on the same machine. The DCE RPC daemon (dced) is used to provide the services of the NCS llbd daemon and is run instead of llbd. To run NCS 1.5.1 applications within a DCE environment, for example, iFOR/LS applications, do the following:

1. Install the NCS and iFOR/LS software if not already on the system.
2. Set up the NCS system manually or use the `netls_config` shell script in the /usr/lib/netls/conf directory.
3. Edit the `netls_first_time` shell script generated by `netls_config`, and comment out the line with the command:

   ```
   startsrc -s llbd
   ```

   to

   ```
   # startsrc -s llbd
   ```

4. Uncomment and change the line that says:

   ```
   /etc/rc.dce rpc
   ```

   to

   ```
   /etc/rc.dce dced
   ```

5. Start the netls_first_time shell script.

10.8 Putting It All Together: Initialization, Routing and Execution

With this section and Figure 72, we try to recapitulate and give the big picture on how all the RPC components work together.

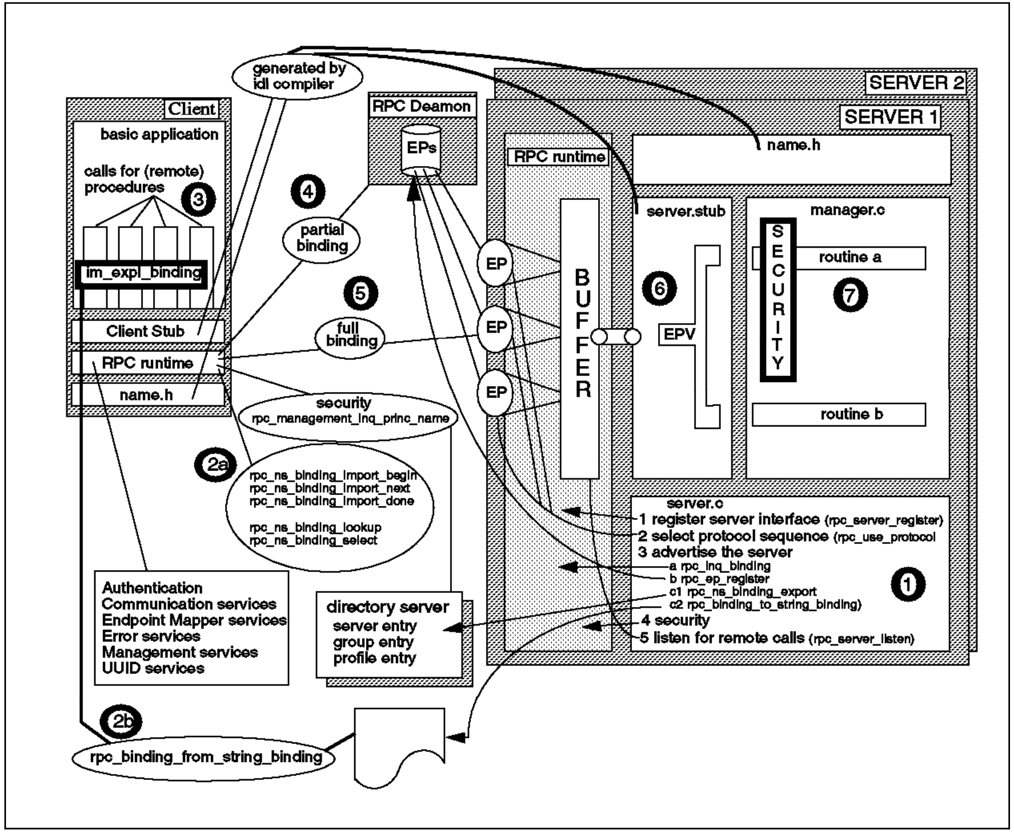


Figure 72. Putting It Together.

The sequence of actions taken to execute a remote procedure call are:

1. Server initializes

   The server part of the application server can optionally create its own login identity. Then it registers its interface(s) (`rpc_server_register`) and the protocol sequence(s) it supports (`rpc_use_protocol`) with the local RPC runtime. The latter step creates the dynamic endpoints (EPs) and binding handles the server will be listening on, one per protocol sequence. The server must query the binding handles from the RPC runtime (`rpc_inq_binding`) in order to be able to register the endpoints with the DCE daemon. This step enables the DCE daemon to route calls to the right EPs. The server then exports its binding information to a CDS server entry (`rpc_ns binding_export`) and creates entries in RPC groups and/or profiles. Another way to make the bindings accessible to servers is to convert them to string bindings and write them to a file. The next step is to optionally set up authenticated RPC. This includes specifying the server principal, authentication level of clients and servers, encryption level, and whether authorization is to performed based on client EPACs. Finally, it initiates listening (`rpc_server_listen`).

2. Client imports a server binding

   If the client looks up information in CDS (2a), it can either use the `rpc_ns_binding_import` or the `rpc_ns_binding_lookup` commands which return binding information in slightly different ways. If the automatic binding method is used, the client′s RPC runtime performs the lookup behind the scenes. For the explicit or implicit binding methods, the client application would have to do this. It would have to analyze and select a suitable binding handle and assign it to a global variable (implicit binding) or use it in every RPC call (explicit binding). If the client uses the explicit or implicit binding method, it can, for instance, also take a string binding and convert it to binding handle (2b).

3. Client issues an RPC call

   After the client has the binding handle, it can add to it the desired security level for the RPC calls. Then it issues an RPC to the server. The client calls the remote procedure, which actually goes to the client stub. The client stub gets the arguments, marshalls them and calls the RPC runtime.

4. Client′s RPC runtime requests an endpoint

   Binding handles obtained from CDS are usually incomplete; they lack an endpoint. The client RPC runtime must contact the endpoint mapper (dced) of the server machine. The endpoint mapper, to which all application servers register their interfaces, searches its database and returns the full binding handle for a (randomly selected) compatible server. If the client specified a fully bound handle with its RPC call, this step would not be necessary.

5. Client′s RPC runtime calls the application server

   The client′s runtime can now make a call to the application server. The client RPC runtime transmits the remote procedure calls (and arguments) to the server′s RPC runtime at the specified endpoint. Remember that an endpoint is just a matter of a protocol sequence. If the server supports multiple interfaces and managers, calls for all of them come to the same endpoint. The server′s RPC runtime receives the request and selects the appropriate manager entry point vector (manager EPV) to handle the request based on the interface, the version and the type UUID requested.

6. Server′s RPC runtime calls the manager routine

   The server stub unmarshalls the arguments and calls the requested procedure.

7. The manager routine executes the call

   The remote procedure begins execution. It extracts the client principal name and its Extended Privilege Attribute Certificates (EPACs) and checks whether the client is authorized to execute the procedure. If the server has implemented an ACL manager, the security information is passed to the ACL manager for evaluation of the permissions. Then the procedure executes and returns the results to the server stub, which marshalls the results and transmits them, via server′s RPC runtime, back to the client.

Chapter 11. Threads

Threads support the creation, management and synchronization of multiple concurrent execution paths within a single process. This provides a great deal of flexibility to application developers in a variety of areas, such as parallel execution, task control, exploitation of multiprocessor machines, and faster task switching. On the other hand, threads introduce considerable, additional complexity.

The DCE core services, and all dependent applications, use threads. This all happens behind the scenes. Customer applications may or may not use threads for their applications.

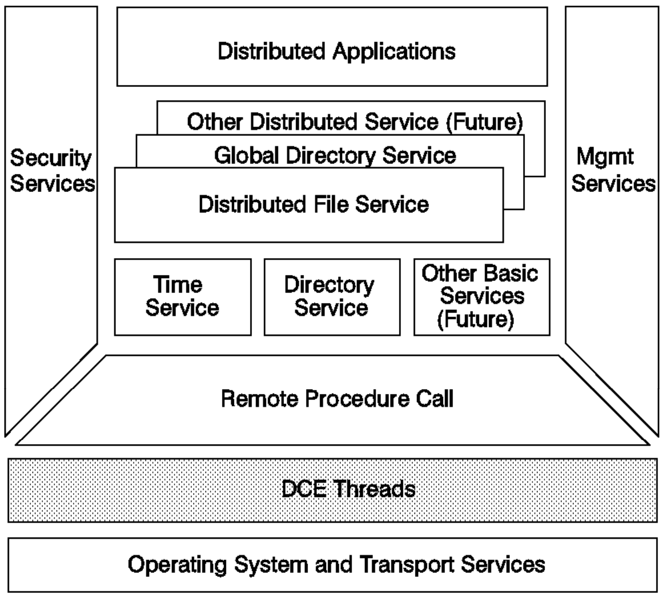


Figure 73. Threads as the Basis for DCE

This chapter discusses the DCE threads implementation and some basic concepts of threads programming. DCE application developers should become familiar with threads programming. DCE application servers work most efficiently when they allow for concurrent execution of client RPC calls, and DCE clients can access multiple servers in parallel only when they use threads. Threads are also needed to perform such things as refreshing tickets or passwords and for implementing application-level timeouts.

DCE users and administrators are not concerned with threads.

11.1 What are Threads?

A thread is the abstraction of a processor. It is a single sequential flow of control within a program. Most computer programs use only one thread of control. Execution of the program proceeds sequentially, and at any given time, only one point in the program is currently executing. It is useful, sometimes, to write a program that contains multiple threads of control for one of the following reasons:

- The application includes parallel algorithms well suited for multiprocessor systems.
- An application accessing slow I/O devices does not want to become blocked.
- The RPC application needs to access multiple servers at the same time.
- An RPC server needs to serve multiple clients.
- The user interfaces can be decoupled from processing.

Most of the preceding structure examples could be implemented on multiple processes, but they have higher creation costs, require more memory and synchronization is more expensive than with threads.

Threads are lightweight processes that share a single address space. Each thread shares all the resources of the originating process, including signal handlers and descriptors. Each thread has its own thread identifier, scheduling policy and priority and the required system resources to support a flow of control.

11.1.1 Multithreading

An environment with multiple concurrent flows of control within a process is said to be a multithreaded environment. This multithreaded environment is desirable for applications requiring multitasking. Software models for multithreaded programming are:

- ***Boss/Worker Model*** — One main thread is performing the boss functions. It assigns tasks to worker threads. When a worker thread has finished a task, it interrupts the boss to get some other jobs done.

  The *Queue Model* is a variation of the Boss/Worker design. Each task is enqueued by the boss thread and when a worker thread is ready, it checks the queue to find the next job it has to run.

- ***Work Crew Model*** — In this model, a task is divided into microtasks that can be run concurrently. The microtasks are run by multiple threads working together to provide the work of the original task.

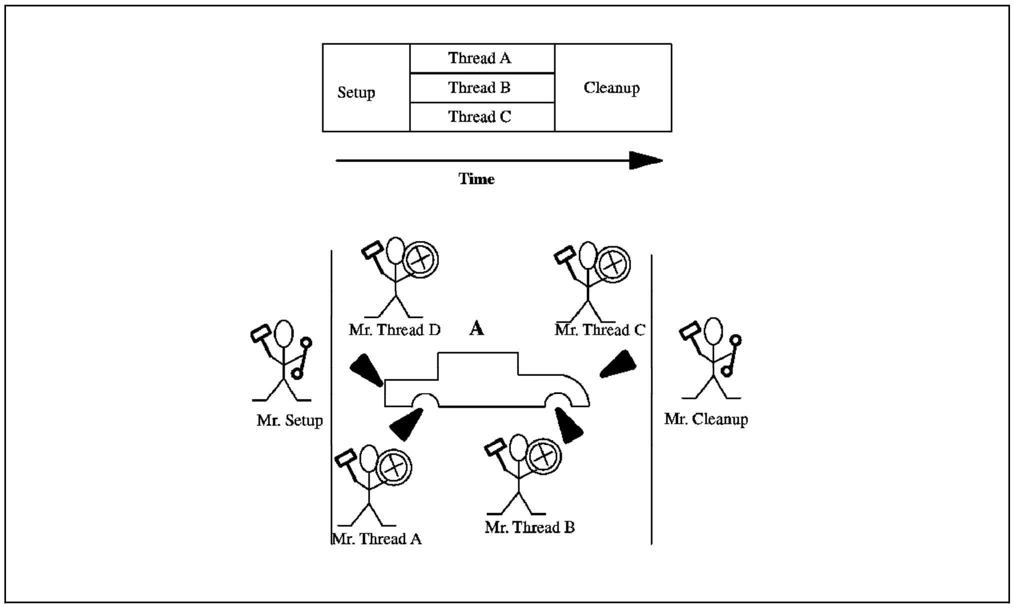


Figure 74. Work Crew Model

An example of this model could be the assembly of the four wheels on a car. Four threads can each assemble a wheel to the car; in this way, you divide the wheels assembly time by a little bit less than four.

- ***Pipelining Model*** — In this model, the task is divided into steps. Each step is a specialized task that can be provided by specialized threads. At each time, the thread can be performing the same task on the *output* of the previous thread. This model can only apply if there is a regular flow of tasks that are always divided into the same steps.

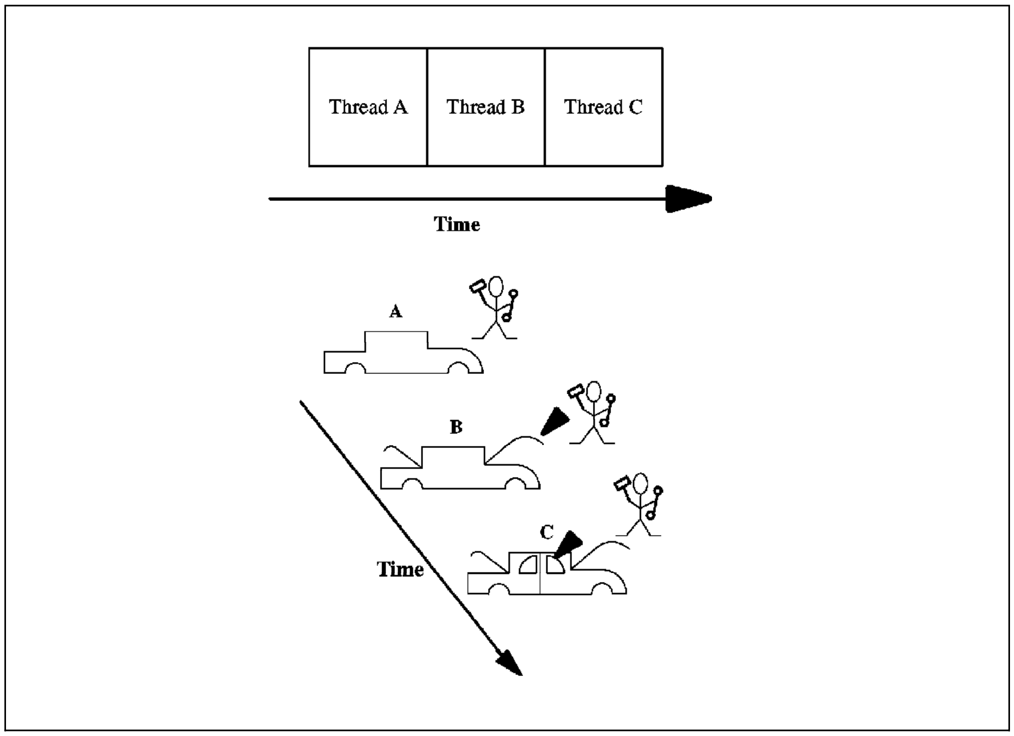


Figure 75. Pipelining Model

We can compare this model with a car assembly line, where each thread is performing the same work on any car that is running on the line.

- ***Combinations of Models*** — Most of the applications will require different types of design models to make the use of threads more efficient. For example, a program could be divided into steps (Pipeline Model), and some of these steps could be designed on the Work Crew Model. This is the case for the assembly line where the wheel assembly is a step of the general pipeline design.

With a thread package, a developer can create several threads within a process. Threads execute concurrently in a single address space. They can run in a user or kernel space. Within a multithreaded process, there are multiple paths of execution at any time. Multiple threads of control allow an application to overlap operations, such as reading from a network connection and writing to a disk file or printing a file while reading and processing user input at the same time.

11.1.2 Benefits of Multiple Threads

Multithreaded programming can bring the following benefits:

- ***Shared resources*** — Multiple threads share a single address space, all open files and other resources.
- ***Exploitation of multiprocessor machines*** — Threads are required to most efficiently use multiprocessor systems.
- ***Performance*** — Threads improve the performance of a program. In multiprocessor machines, threads can concurrently run on separate processors. However, multiple threads also improve program performance on single-processor systems by overlapping slow operations with computational operations.
- ***Potential simplicity*** — Multiple threads might reduce the complexity of some applications that are suited for threads.

On the other hand, this new concept may also introduce some additional complexity in managing concurrent access to shared resources.

11.1.3 Implementation Models

There are at least two strategies for implementation of a thread system. Put simply, the individual threads may be known (and supported) by the operating system, or they may exist strictly at the user level.

When the threads only exist at the user level, we have an N:1 model; this means that all threads visible to the process are mapped onto a single kernel thread. This is illustrated as model (a) in Figure 76 on page 213. AIX Version 3 works that way.

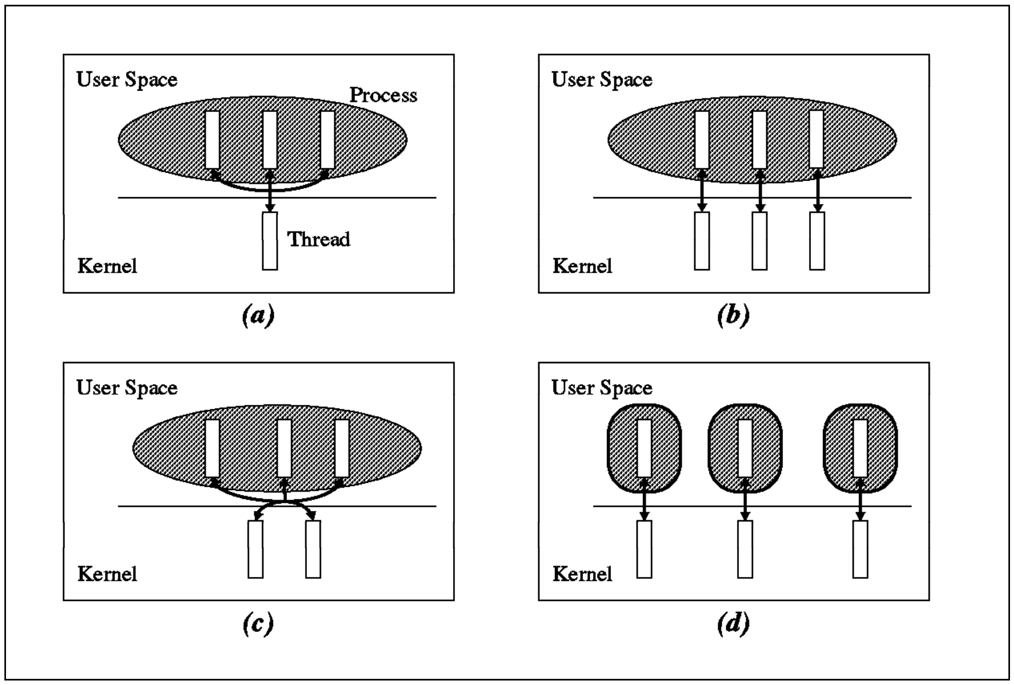


Figure 76. Threads Implementation Models

Model (b) in Figure 76 implements a 1:1 model; for every thread visible to an application, there is a corresponding kernel thread. This is how AIX Version 4, which is based on the OSF/1 libpthreads, and OS/2 work.

It is also possible to have an M:N threads model, like model (c) in the picture, where M user threads are multiplexed on N kernel threads. This is, however, not supported by AIX or OS/2. Model (d) is the traditional UNIX process environment, where the process is a single thread implementation.

An issue is whether or not threads can take advantage of multiple (real) processors.

11.2 DCE Threads Implementation

DCE Threads is based upon Digital′s implementation of Concert Multithread Architecture (CMA). The DCE threads component is based on the threads interface specified by POSIX in 1003.4a Draft 4, called *pthreads* interface. It is designed as a user-level thread package that can run on operating systems that do not support threads in their kernel.

If the operating system does not support threads in its kernel, the threads are running in (non-privileged) user mode. The kernel is then not aware of threads running in a process — it can only see (and dispatch) the process as a whole. The problem is that one thread could put the entire process into a wait state, thereby making all other threads also wait. Programmers have to be aware of this situation if they use threads. To avoid blocking the process with a thread, they should either use only calls of thread-safe, reentrant libraries, use asynchronous I/O calls or write their applications in a way that one server is only talking to one client at a time and vice versa.

If a system has an alternative implementation of POSIX compatible threads, the DCE thread calls may be mapped directly to kernel threads, and the DCE threads library just has a mapping function. In this case, the intermediary library might have to adapt different levels of threads implementation between the Draft 4 threads defined by DCE and another level implemented by the kernel threads. For example, the AIX Version 4.1 libpthreads implementation is based on the newer POSIX 1003.4a Draft 7 specification.

Routines implemented by DCE Threads that are not specified by Draft 4 of the POSIX 1003.4a standard are indicated by an *_np* suffix on the name. These routines are not portable.

11.3 Threads Basics

This section describes the basic concepts behind DCE threads. For detailed information on the multithreading routines referenced in this section, see the chapters on threads in the *DCE Application Development Guide Core Components* for AIX or OS/2.

11.3.1 Threads States and Control Operations

A thread is created using the `pthread_create()` routine, which returns to its caller a unique identifier (handle) for that thread. Like a traditional process, an executing thread is then subject to state transitions until it terminates. As illustrated in Figure 77, a thread is in one of the following states:

- ***Waiting*** — The thread is not eligible to execute because it is synchronizing with another thread or with an external event, such as the completion of an I/O.
- ***Ready*** — The thread is eligible to be executed by a processor.
- ***Running*** — The thread is currently being executed by a processor.
- ***Terminated*** — The thread has completed all of its work.

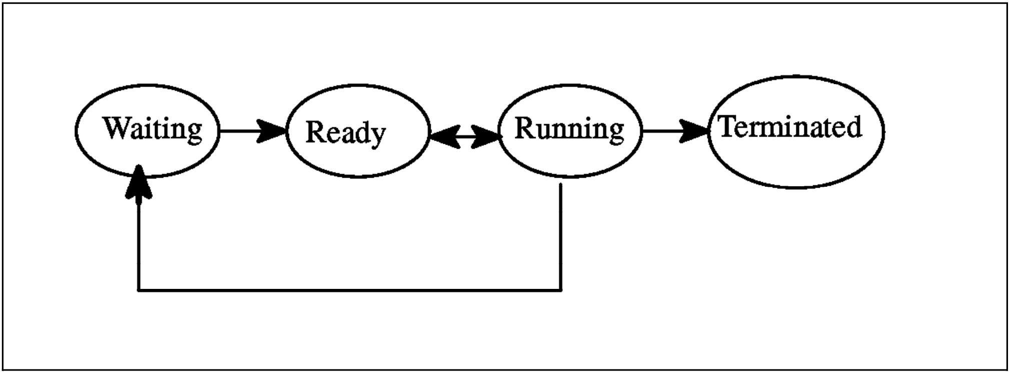


Figure 77. Thread State Transitions

Once a thread is executing, there are several operations that can be performed on it, such as:

- `pthread_join()` — Another thread suspends its execution (waits) until this thread terminates. The caller must know the unique identifier (handle) for the thread it wants to wait for. If multiple threads call this routine and specify the same thread, all threads resume execution when the specified thread terminates. Calling the `pthread_join()` routine on the current thread causes a deadlock.

- pthread_cancel() — Another thread may call this routine to request that a running thread terminate itself.
- pthread_detach() — After a thread terminates, it continues to live and can be joined to by others. Another thread (or the main thread) must call this routine to clean up the allocated resources of the terminated thread. If called before termination, the thread will be cleaned up right after its termination, and no other thread can join to it or cancel it.
- pthread_exit() — A thread can specify multiple points to regularly terminate in addition to just running up to the last statement.

More details and operations can be found in the *DCE Application Development Guide*.

11.3.2 Thread Attributes

Attributes can be assigned to threads, mutexes and condition variables. If such an object is to be created with attribute values different from the default values, a so-called ***attributes object*** must be created and provided to the routine that creates the object. An attributes object is basically a collection of attributes that will be assigned as a whole. Internally, it is a data structure that is referenced by an object handle (pointer).

The pthread_attr_create() call is used to create a ***thread attributes object***, which may contain the following attributes:

- Scheduling Policy Attribute
- Scheduling Priority Attribute
- Inherit Scheduling Attribute
- Stacksize Attribute

When creating a new thread, such a thread attributes object can be provided to the pthread_create() call to overwrite the default values for thread attributes.

The pthread_attr_set..() routines can be used to change attributes in an already established attributes object, and attributes of executing threads can be altered by one of the pthread_set..() routines. For example, a thread is created with a thread attributes object that contains a priority. This priority was set up with a pthread_attr_create() call and may be altered by a pthread_attr_setprio() routine before the thread is created. Once the thread is running, its priority can only be changed with a pthread_setprio() routine.

Similarly to thread attributes, you can create a ***mutex attributes object*** with a pthread_mutexattr_create() call. To overwrite default values for mutex attributes, this object must then be specified in the pthread_mutex_init() call that creates a mutex. The mutex *type* attribute specifies whether a mutex is fast, recursive or non-recursive (see 11.3.4.1, “Mutexes” on page 220).

No attributes are currently defined that affect condition variables. The stack size attribute is the minimum size (in bytes) of the memory required for a thread's stack. The default value is machine dependent. The attribute may be set with the pthread_attr_setstacksize() routine. There is currently no support for extending the size of a stack when overflow is encountered.

11.3.3 Threads Scheduling

If there are fewer available processors than the number of threads to be run, some decision must be made as to which thread runs first. This is analogous to the scheduling of processes by the operating system on a timesharing system, except that the threads scheduling is visible to and controllable by the application programmer. DCE threads scheduling is built on two basic, interacting mechanisms:

- Scheduling priorities
- Scheduling policies

Each thread has a scheduling priority associated with it. Threads with a higher priority have precedence over threads with a lower priority when scheduling decisions are made. The exact treatment of threads of different priorities depends on the scheduling policy they are running under.

Note: POSIX specifies that scheduling is optional. So, systems using their own threads implementations may miss the functionality provided by DCE threads.

11.3.3.1 User-Level Scheduling

One approach to designing a threads system is to implement them strictly at the user level. On a UNIX system, the strategy is for the entire program to run within a process. The operating system provides its own, single thread; the user-level threads system must multiplex this operating system thread among any number of user threads. So, scheduling threads is done within the process, whereas the operating system schedules the process as a whole. This is illustrated in Figure 78 on page 217.

A major advantage of this approach is that most operations on threads do not require calls to the operating system and thus are relatively inexpensive. A potential disadvantage is that a blocking system call will more often block the entire process and not only the threads that have made the call. DCE threads on AIX 3.2 are based on this approach, the AIX scheduler dispatches processes only.

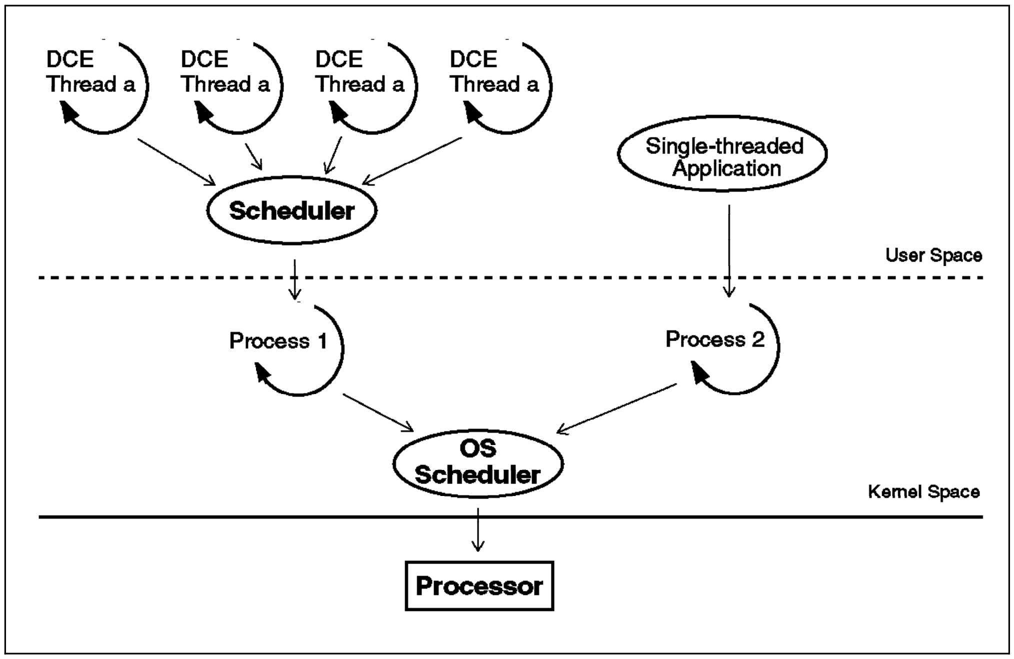


Figure 78. User-Level Scheduling

11.3.3.2 Kernel-Level Scheduling

In this case, threads are supported by the operating system (this is the case of OSF/1, OS/2 and AIX 4.1). Thus, only one scheduler is needed, the one in the operating system kernel. Threads are the dispatchable unit for the OS scheduler. On these operating systems, all system calls can be executed by threads without blocking the entire process.

In AIX Version 4.1 and OS/2, the implementation of threads is 1:1. This means that for every user space thread there will be one kernel space thread to support it. They are permanently bound to each other until the thread terminates. Architecturally, this type of implementation produces user threads that are said to be *system scope threads*. All system scope threads contend with each other for resources. This is illustrated in Figure 79 on page 218.

The POSIX 1003.4a Draft 7 specification allows for an alternative implementation of threads. In particular, it allows the M:N implementation, where, in addition to system scope threads, there can be user threads that are not permanently attached to kernel threads. These threads are called *process scope threads*. This allows multiple user threads to be scheduled and multiplexed onto a smaller number of kernel threads. This is done in user space, as part of the threads library function, thereby reducing demand on system resources. This sort of implementation is particularly advantageous for applications that have large numbers of threads most of which are waiting most of the time. It can also bring performance improvements for thread creation and deletion because the OS threads are recycled. IBM MVS OpenEdition threads work this way.

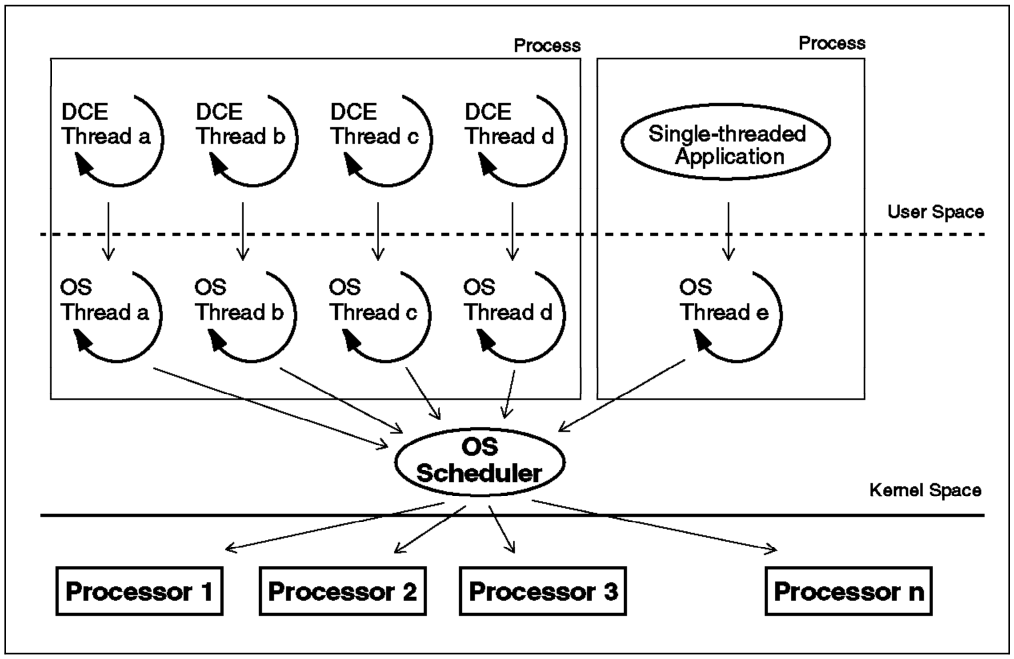


Figure 79. Kernel-Level Scheduling

11.3.3.3 Scheduling Priorities and Policies

For each scheduling policy, a maximum and minimum priority value is defined. When setting the ***scheduling priority*** for a thread with the `pthread_attr_create()`, the `pthread_setprio()` or the `pthread_attr_setprio()` routines, a priority value must be specified within the range defined for the chosen policy.

The ***scheduling policy*** determines how the priorities are interpreted and used to dispatch the threads. The `pthread_attr_create()` or `pthread_attr_setsched()` calls are used to set the scheduling policy *and* priority on attributes objects which are used to create new threads. See 11.3.2, “Thread Attributes” on page 215 for more information on attributes. The `pthread_setscheduler()` routines are used to change the policy *and* priority of an active thread.

There are three policies according to which the prioritized threads are dispatched. For the explanations of these policies, let′s assume we have four threads (A, B, C, D), and we have assigned to them three different priorities (min, mid, max), which are within the range of valid priorities:

A	min
B	mid
C	mid
D	max

We assume that all waiting threads are ready to execute when the current thread waits or terminates and that no higher priority thread is awakened while a thread is executing (during the flow). This is what happens within each policy:

- ***FIFO (First In / First Out)*** or `SCHED_FIFO` — This is a non-preemptive scheduling mechanism. A thread created with `SCHED_FIFO` will run at a fixed priority and will not be timesliced. The thread in the highest priority category is scheduled first and is allowed to run until it voluntarily relinquishes by blocking or yielding. We get the following thread scheduling sequence:

```
D --> B --> C --> A
```

When two waiting threads have the same priority, the one that has to wait longer is scheduled. That is why B is before C. A SCHED_FIFO thread with a high enough priority could monopolize the processor.

- ***Round Robin*** or SCHED_RR — The highest-priority thread runs until it blocks; however, threads of equal priority, if that priority is the highest among other threads, are timesliced. The timeslicing is a mechanism that ensures that every thread is allowed time to execute by preempting running threads at fixed intervals. We get the following sequence:

  ```
  D --> B --> C --> B --> C --> A
  ```

  Round Robin threads are like FIFO threads, except that they are timesliced. Thread D executes until it waits or terminates. Then, execution of threads B and C is timesliced because they both have the same priority. Finally, thread A executes.

- ***Default*** or SCHED_OTHER — Each thread is given turns running by timeslicing. Higher priority threads are given longer periods of time to run, but even the lowest priority thread eventually has a chance to run. We get the following sequence:

  ```
  D --> B --> C --> D --> A --> B --> C --> . . . .
  ```

  This is much like the normal AIX scheduling, where priority degrades with CPU usage. Because low-priority threads eventually run, the default scheduling policy protects against the problem of *priority inversion* (see 11.4.2, “Potential Problems with Multithreaded Programming” on page 226).

- ***Foreground / Background*** or SCHED_FG_NP, SCHED_BG_NP — These policies are basically the same as the default policy. The SCHED_FG_NP is just another name for SCHED_OTHER, whereas threads with SCHED_BG_NP only run when no other threads are ready to run. The _NP routines are not POSIX-conformant.

If the DCE threads component uses kernel scheduling, policies can only be set by a user with root authority.

By default, a thread inherits its attributes, including its scheduling priority, from the thread that creates it. Whether a newly created thread inherits the scheduling attributes of the creating thread or not is determined by the inherit scheduling attributes that can be set by the pthread_attr_setinheritsched() call.

11.3.4 Threads Synchronization

Threads can communicate through shared variables: One thread sets a variable that another thread later reads. However, concurrent access to shared data might lead to conflicts and data corruption. Also, threads can depend on each other, but due to scheduling policies and priorities, one might terminate before the other expects it. For both cases, a synchronization mechanism must be in place.

DCE threads contain an implementation of the POSIX 1003.4a required synchronization routines, which provide three features:

1. ***Mutexes*** — Stands for *mutual exclusion* and ensures that no more than one thread may have access at any one time to certain data structures.
2. ***Condition variables*** — Used in conjunction with mutexes to provide a more sophisticated form of synchronization.

3. ***Join routine*** — Allows a thread to wait for another, specific thread to complete its execution. When the second thread has finished, the first thread unblocks and continues its execution.

Unlike mutexes and condition variables, the join routine (`pthread_join()`) is not associated with any particular shared data. While the `pthread_join()` call is explained in 11.3.1, "Threads States and Control Operations" on page 214, we want to explain the mutex and the condition variable in the following subsections and give a programming example thereafter.

11.3.4.1 Mutexes

A mutual exclusion or (*mutex*) is an object that multiple threads use to ensure the integrity of a shared resource that they access. It can have two states: locked and unlocked. For each piece of shared data, all threads accessing that data must use the same mutex to lock the data before accessing it with `pthread_mutex_lock()` and unlock it with `pthread_mutex_unlock()` when finished.

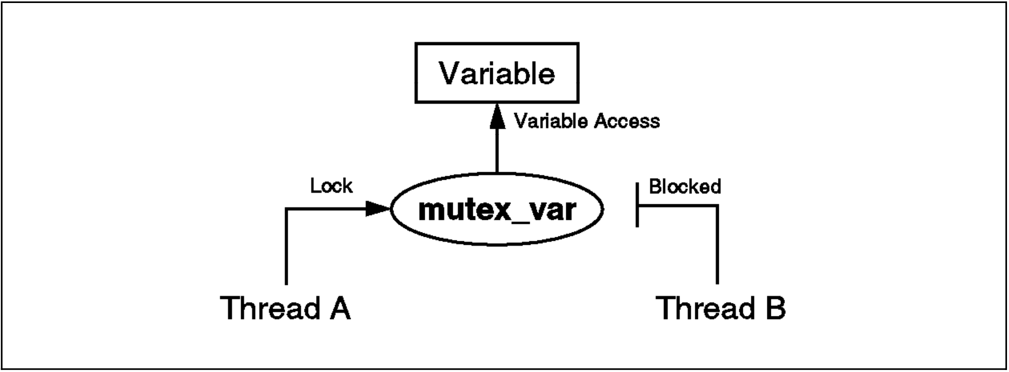


Figure 80. Only One Thread Locks a Mutex

When a thread, B in Figure 80, requests the lock of a mutex that is already locked by another thread, A, thread B′s lock is blocked. When thread A that owns the block finishes with the shared data and unlocks the mutex, thread B is unblocked and gains control of the mutex. Thread B can also call `pthread_mutex_trylock()`, which would not block and would allow it to continue processing.

Each mutex must be initialized with the `pthread_mutex_init()` routine. This routine allows one to specify one of three following mutex types:

1. ***Fast mutex*** (default) — Can be locked and unlocked more rapidly than other mutexes. It is the most efficient form of mutex. However, it provides little error checking, and if a thread calls the lock a second time before unlocking it, the thread gets blocked and deadlocks itself.
2. ***Recursive mutex*** — Thanks to a recursion count, this mutex can be locked several times by the thread that already holds the mutex without causing a deadlock. The thread has to call the `pthread_mutex_unlock()` routine the same number of times it has called the `pthread_mutex_lock()` before another thread can lock the mutex.
3. ***Non-recursive mutex*** — Like with the fast mutex, a thread can only hold one lock on a mutex at any point in time. However, if it tries to lock the mutex again without first unlocking it, the thread receives an error. Thus, non-recursive mutexes are more informative than a fast mutex. This kind of mutex is very useful during application debugging.

Note: The concept of different types of mutexes is not required in POSIX 1003.4a, but is explicitly permitted.

11.3.4.2 Condition Variables

A condition variable is a special type of shared data used for explicit communications among threads. A condition variable allows a thread to block its own execution until some shared data reaches a particular state. Cooperating threads check the shared data and wait on the condition variable.

Like any other shared data, the condition variable is protected by a mutex. However, if the synchronization were based on a regular shared variable, the thread would have to go into a loop that locks the mutex, checks the variable, unlocks the mutex, and waits for awhile before it goes through the same cycle over and over. What is special about the shared condition variable is that it provides a single `pthread_cond_wait()` call to the threads kernel that internally unlocks the mutex and waits until the variable reaches the desired value or state.

For example, one thread in a program produces work-to-do packets, and another thread consumes these packets. If the work queue is empty when the consumer thread checks it, the thread waits on a work-to-do condition variable. When the producer threads puts a packet on the queue, it signals the work-to-do condition variable. Note that although the condition variable is used for explicit communication among threads, the communication is anonymous.

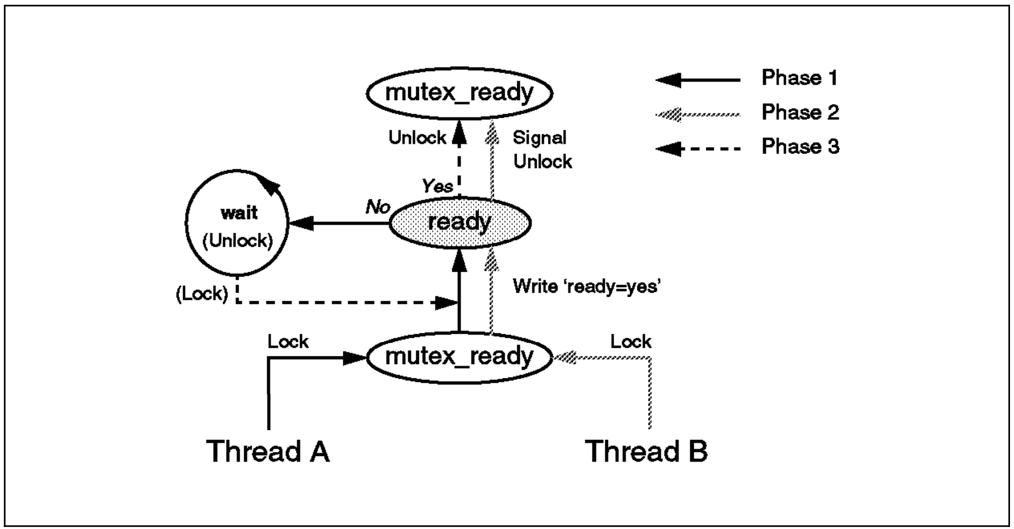


Figure 81. Synchronization via a Condition Variable

In Figure 81, thread A may need to wait for a thread B to finish a task X before thread A proceeds to execute task Y. A condition variable *ready* is defined that needs to be set to *yes* by thread B, before thread A continues. Thread A needs to lock the mutex and checks the condition variable a first time. It unlocks the mutex and continues, if *ready=yes*; otherwise, it calls `pthread_cond_wait()`. Much like the `select()` system call in UNIX, this call blocks until the variable is set to *yes*.

The `pthread_cond_wait()` call passes the variable and the mutex to the threads kernel which unlocks the mutex to enable write operation to another thread. Thread B can then lock the mutex, set the variable, unlock the mutex, and awaken waiting threads with one of the following calls:

- `pthread_cond_signal()` awakes one thread. If more than one thread is waiting, the one with the highest priority is signalled.
- `pthread_cond_broadcast()` awakes all waiting threads.

When thread A is awakened, the mutex is automatically locked again. Thread A checks the condition and, depending on whether it is what it expected, continues or decides to continue to wait.

A condition variable is initialized with a `pthread_cond_init()` call and eventually cleaned up with a `pthread_cond_destroy()` command. The waiting thread can also call a time-limited wait by using the `pthread_cond_timedwait()` call.

11.3.4.3 Thread Programming Example

We have a simple code that maintains the queue of a resource — one thread adds queue elements, another thread dequeues and processes elements.

The resource in our example is a structure with the following members:

- A count of the number of items in the list
- A mutex to ensure that only one thread at a time is accessing the header
- A condition variable used to wait for items to be added to the list
- A pointer to a list of elements (the queue)

The code to acquire a queue element is:

```
acquire_resource(resource) {
   /* gain exclusive access to the header by blocking its mutex */
   ret=pthread_mutex_lock(resource.mutex);
   /* check whether there are any items on the queue */
   while (resource.count == 0)
     /* waiting for an item to be added to the list by using the condition variable */
     /* A while loop is better than an if statement, because it will check again*/
     ret = pthread_cond_wait(resource.cond, resource.mutex);

   /* Now an item is in the queue; call a routine that properly dequeues it */
   acquired_resource = dequeue(resource.queue);
   resource.count --;

   /* Release mutex and and return pointer to dequeued element */
   ret = pthread_mutex_unlock(resource.mutex);
   return (acquired_resource);
   }
```

The code to enqueue an element is:

```
release_resource(resource) {
   /* Lock the resource */
   ret=pthread_mutex_lock(resource.mutex);
   /* Queue an element and increase the count */
   enqueue(resource.queue, resource);
   resource.count ++;
   /* Release the mutex lock and signal a dequeuing thread */
   ret=pthread_mutex_unlock(resource.mutex);
   ret=pthread_cond_signal(resource.cond);
 }
```

11.4 More Advanced General Threads Topics

This section covers, still at a high level, some more specialized topics, such as error handling and potential problems in relation with threads programming. See also 11.5, “More Advanced Threads Topics in UNIX” on page 227 for UNIX-specific topics, such as signals and forking processes.

11.4.1 Error Handling

An exception is an abnormal condition that can occur during program execution. Error checking and error handling is extremely important in a multithreaded environment. The conditions in which a thread is executing can change at any time due to the activity of other threads executing concurrently on the same address space.

DCE threads provides the following two ways to obtain information about the results of a threads call:

- The routine ***returns a status value*** to the thread. This method is specified by the POSIX 1003.4a (pthreads) Draft standard. Errors are reported to the thread by setting the external ***errno*** variable to an error code and returning a function value of -1.
- The routine ***raises an exception***. This is provided by the exception-returning interface, which is a DCE extension to the basic POSIX functionality (and is not standard). When an exception is raised by any level of nested routines, program execution jumps to a predefined section in the code and continues from there. Exception error handling must be explicitly enabled. See 11.4.1.2, “Enabling Exceptions” on page 224 for more details.

  Note: On IBM DCE Version 2.1 for AIX, the exception error handling is only available through the DCE Pthreads Compatibility library.

Before you write a multithreaded program, you must choose one of these two methods of receiving error status. They cannot be used together in the same code module.

11.4.1.1 Exceptions

Exceptions represent a deviation in the normal flow of control of a thread, typically caused by an error. These errors can be:

- Hardware errors (for example, divide-by-zero)
- Operating-system errors (for example, invalid argument on system call)
- User-defined errors

User-level errors are managed by return codes. In UNIX, the first two categories normally result in a signal sent to the faulting process. See 11.5.1, “Signals” on page 227 for further explanations. A few signals are converted into exceptions, and the rest must be handled by a thread which could, for instance, raise an exception. Here is the syntax of exception handling:

```
TRY
   try_block
[CATCH (exception_name)
       handler_block   ] ...
[CATCH_ALL
       handler_block   ]
ENDTRY
```

The try_block and handler_block are a sequence of statements. If an exception is raised in the try_block with a statement RAISE(exception_name), the program jumps to the inner-most exception block defined by the CATCH(exception_name) or, if undefined, to the CATCH_ALL block. If the code within a handler block does not fully handle an exception, it should call RERAISE to further propagate the exception. If it does reraise, then the next handler that might exist higher up in a nested hierarchy is called. If it does not reraise, propagation stops, and execution is resumed after the ENDTRY statement. The thread terminates if the exception remains unhandled.

Instead of using CATCH/CATCH_ALL, you can also use a common epilogue, the FINALLY statement, as follows:

```
TRY try_block
[FINALLY final_block]
ENDTRY
```

The statements defined in a final block are executed on success or after an exception has been raised. In the exception case, propagation of the exception is resumed after execution of the final block, and an implicit RERAISE is performed.

11.4.1.2 Enabling Exceptions

To use the exception returning interface, the program must include the dce/pthread_exc.h file instead of the pthread.h file, which is used to obtain *errno* values. For DOS Windows, we should include *pthreadx.h*.

11.4.1.3 Special Considerations for the Main Thread

On OS/2, to install the exception handler for the main thread, we have to invoke the pthread_inst_exception_handler() macro. The exception handler is automatically installed on threads created with the pthread_create() call. The macro should be the first executable statement of the main(). The macro defines some local variables. The pthread_dinst_exception_handler() macro has to be called as the last executable statement just before the *return*.

On AIX, we can invoke pthread_signal_to_cancel_np(). This is a DCE threads API call which specifies a thread to be cancelled when a valid signal is received by the process. We set the signal mask so that the SIGINT and SIGTERM signal cancel the main thread.

11.4.1.4 Example (Banking)

In this example, you can approach the notion of exception and how it is handled. The following code can be part of a client transaction calling a withdraw() and deposit() function. These two functions are enabled to raise exceptions. The main transaction code contains the TRY-CATCH-ENDTRY clauses:

```
TRY {
      withdraw(accountA,amount)
      .               .         /* this part defines the code over which */
      .               .         /* the exception are caught :           */
      .               .         /*        the exception-handling region */
      deposit(accountB,amount)
}
CATCH (red) {                    /* example of catching exception       */
      warning(accountA);         /* and specifying the recovery action */
      printf("ALERT trying to get more money than there is\n");
      RERAISE;
```

```
  }
  CATCH (unknown_acct) {
        deposit(temp_acct, amount);
        warning_action(accountB);
        RERAISE;
  }
    .     .
    .     .
    .     .
  CATCH_ALL {             /* absorbs the rest of the exception that have  */
        big_trouble();     /* not been caught in the other CATCH clauses   */
        RERAISE;
  }
  ENDTRY
```

The functions that raise the exceptions can be as shown below:

```
  withdraw(account,amount) {
    if (amount > total)
      RAISE(red);
    total -= amount;
  }

  deposit(account,amount) {
    if (!exist(account))
      RAISE(unknown_acct);
  }
```

If any other exception propagates out of withdraw, `big_trouble()` will be executed. In either situation, the propagation of the exception resumes because of the `RERAISE` statement and eventually terminates the thread if not handled by an upper-level handler.

11.4.1.5 DCE Threads Exception Handling on DOS Windows

IBM DCE for DOS Windows has extensive error and exception handling. System error conditions can be displayed to the user and recorded in an error log. Exceptions are managed through the DCE exception handling facility. The implementation uses a set of preprocessor macros and calls interface in the DCE DLL to maintain a stack of exception handlers in exactly the same manner as UNIX DCE.

In IBM DCE for DOS Windows, there are no floating point or arithmetic exception handlers. They can be raised by translating the errors into the corresponding DCE exception.

Some exceptions that can be handled in UNIX cannot be trapped in DOS Windows applications, including most forms of memory addressing violations. The DOS Windows kernel will usually report an unrecoverable application error and kill the application. This is a restriction of DOS Windows 3.0 and 3.1.

Servers running on UNIX or other platforms can return exceptions to a DCE client on Windows that are not normally recoverable exceptions to a DOS Windows program. The client should be prepared to deal with such exceptions. The DCE API has two interfaces: *exception-handling* and *exception-returning*. Both interfaces are defined in the same *pthreadx.h* header file.

11.4.2 Potential Problems with Multithreaded Programming

Although thread programming has many advantages over traditional processes (performance, shared resources), it also introduces new issues in other areas, such as the following:

- ***Potential Complexity*** — The level of expertise required for designing, coding and maintaining multithreaded programs may be higher than most single-threaded programs. A multithreaded program may need shared access to resources, mutexes and condition variables. Weigh the potential benefits against the complexity and its associated risks.
- ***Non-Reentrant Software*** — Thread-reentrant code is code that works properly while multiple threads execute it concurrently. Thread-reentrant code is thread-safe, but thread-safe code may not be thread-reentrant. When you need to call non-reentrant code, you need to *globally* lock its use before you call it. Consult the *DCE Application Development Guide* with the search terms *global lock* mechanism or *thread-specific data interface*.
- ***Priority Inversion*** — Priority inversion occurs when interaction among three or more threads blocks the highest-priority thread from executing. For example, a high-priority thread waits for a resource locked by a low-priority thread, and the low-priority thread waits while a middle-priority thread executes. So, the middle-priority thread executes while the high-priority thread is made to wait.

  To avoid priority inversion, associate a priority with each resource, and force the thread using that object to first raise its priority to that associated with that object. Or use the default scheduling policy (SCHED_OTHER) that prevents priority inversion.
- ***Race Conditions*** — A race condition occurs when two or more threads perform an operation on a shared variable and are interrupted before they end the operation. An example is that thread A reads the variable and is interrupted by a higher-priority thread or the timeslicing mechanism. Thread B changes the value of the variable, and when thread A is running again, it goes from the old value of the variable. The result will be inconsistent and unpredictable.

  To avoid race conditions, we have to ensure that any variables modified by more than one thread have a mutex (see 11.3.4.1, “Mutexes” on page 220) associated with it.
- ***Deadlocks*** — A deadlock occurs when one or more threads are permanently blocked from executing because each thread waits on a resource held by another thread in the deadlock. A thread can also deadlock on itself.

  To avoid a deadlock, use mutexes associated with a sequence number and lock them in sequence, or use a *recursive mutex* if threads need to lock the same mutex more than once before unlocking it.
- ***Blocking Calls*** — On a system that does not have support for kernel threads, certain system or library calls may cause an entire process to block while waiting for the call to complete. As a result, all other threads stop executing. DCE threads provide jacket routines that make certain system calls thread-synchronous. See 11.5.2, “Jacket Routines for UNIX System Calls” on page 228 below for more explanations.
- ***I/O Handling*** — See *Blocking Calls* above.

11.5 More Advanced Threads Topics in UNIX

This section covers, at a high level, some more specialized programming topics present in UNIX environments, such as signals and forking processes.

11.5.1 Signals

Signals are a traditional way of process communication in the UNIX environment; they are a way of notifying a process that a certain event has occurred. Signals are a single mechanism for dealing with two kinds of events:

- An ***exception*** is a result of an event that occurs inside a process and is delivered synchronously with respect to that event. It is also called a ***synchronous signal*** and is typically an indication of a problem.
- An ***interrupt*** is a result of an event that is external to the process, such as a `kill` command or a user pressing Ctrl-c at their terminal. This is considered an ***asynchronous signal***.

There is very little distinction between the two types of signals within a single-threaded UNIX process. One may establish a handler for a signal to catch it. When the signal is raised, the handler is called. Both types of signals can be terminating or nonterminating. Terminating signals terminate the process if the user (programmer) does not catch it.

In a multithreaded environment, it is more complicated. The problem is, when a signal is sent to a process, we have to choose which thread should deal with the signal. Indeed, signal behavior is one of the largest and most debated differences between POSIX 1003.4a Draft 4 and Draft 7. The original DCE pthreads library, including signals, is based on Draft 4. The following is the behavior of the two types of signals, and how they can be handled:

- ***Synchronous Signals*** are delivered to the faulting thread. To handle them, signal handlers must be established on a *per-thread* basis using the `sigaction()` call. If not handled, the default action is usually to dump a core image and terminate. Synchronous signals cannot be waited for by using `sigwait()` because a thread can only trigger this type of signal, such as a segmentation violation, when it is running and not waiting.
- ***Asynchronous Signals*** are directed to the *process* where the DCE Threads catches and handles them by means of jacket routines. This type of signal awakes all threads that called `sigwait()` to wait for that particular signal, and the signal is dismissed (ignored). If there is no thread waiting for a signal, it is considered unhandled when it arrives, and the default action takes place, which usually means terminating the process. A way to handle these signals is to set up one thread per signal to wait for it and act upon it.

The use of UNIX signals in a multithreaded environment is discouraged. By using mutexes and signals on condition variables, you can avoid the use of UNIX signals.

AIX Specialities

AIX DCE is different from OSF DCE in that it allows signal handlers to be set up also for asynchronous signals with the `sigaction()` call.

The DCE Pthreads compatibility library implemented by AIX Version 4.1 maintains the signal semantics of POSIX 1003.4a Draft 7, which is different from that of Draft 4:

- In Draft 7, ***all signal handlers*** are ***per process*** and executed in the context of the thread that is running at the time of signal delivery. If another thread establishes a handler for an already established handler, it overwrites the previous setting. Even if your application compiles without any errors, the pthreads signal behavior may be incorrect. The synchronous signals could be handled per thread in Draft 4.
- In Draft 7, ***signal masks***, used to block signals, are ***per thread***. When a signal is blocked, it cannot delivered to a thread and it is marked pending until the signal mask bit is turned off. In Draft 4, the signal mask was process-wide.

11.5.2 Jacket Routines for UNIX System Calls

DCE threads provide jacket routines for a number of UNIX system calls. Threads call the jacket routine instead of the UNIX system service; this allows DCE threads to take action on behalf of the thread before or after calling the system service. For example, the jacket routine ensures that only one thread calls any particular service at a time to avoid problems with system calls that are non-reentrant and would otherwise block the process.

Jacket routines are provided for the following system calls:

- I/O calls — Examples of jacketed system calls are `read()`, `write()`, `open()`, `socket()`, `send()`, `recv()`.
- `fork()` — This call creates another process as an exact clone. The `atfork()` routine allows you to define actions to clean-up the thread environment right before and after the fork. See 11.5.3, “Calling fork() in a Multithreaded Environment” on page 229.
- `sigaction()` — This call allows you to set up a signal handler. The jacket allows each individual thread to set up its own handlers. Note that in AIX DCE 2.1, signal handlers are only process-wide, as explained in 11.5.1, “Signals” on page 227.

Jackets are not provided for any other UNIX system calls or for any of the C runtime library services, such as `wait()`, `sigpause()`, `msgsnd()`, `msgrcv()` and `semop()`. If a thread makes a call to any non-jacketed, blocking system call, the thread, and with it the whole process, are prevented from executing.

The jacket routine ensures that only the calling thread is blocked and that the process remains available to execute other threads. A list of jacket routines can be found in the /usr/include/dce/cma_ux.h file.

You do not have to rename your system calls to take advantage of the jacket routines. Macros put the jacket routines into place when you compile your program; these macros rename the jacketed system calls to the name of the DCE threads jacket routine. Thus, a reference to the DCE threads jacket routine is compiled into your code instead of a reference to the system call. When the code is executed, it calls the jacket routine, which then calls the system on your code′s behalf.

11.5.3 Calling fork() in a Multithreaded Environment

The fork() system call creates an exact duplicate of the address space from which it is called, resulting in two address spaces executing the same code. Problems can occur if the forking address space has multiple threads executing at the same time of the fork(). When multithreading is a result of library invocation, threads are not necessarily aware of each other′s threads.

POSIX defines the behavior of fork() in the presence of threads to propagate only the forking thread and to eliminate other threads. No cancels are sent and no handlers are run. All other portions of the address space are cloned, including all mutex states. If the other threads have a mutex locked, the mutex will be locked in the child process, but the lock owner will not exist to unlock it. Therefore, the resource protected by the lock will be permanently unavailable. If your code does not attempt to lock a mutex that could be locked by another thread at the time of the fork(), your code will be safe.

The jacketed fork() call allows you to use the atfork() call to set up routines that will run at the following times:

- Prior to the fork() in the parent process
- After the fork() in the child process
- After the fork() in the parent process

With these routines, you can lock all mutexes before the fork() and unlock them after the fork(). Another solution is to save state information of the threads, terminate them before the fork(), and reinstate them as necessary afterwards.

11.6 Platform-Specific Implementation

This section explains the minor differences among implementations of DCE threads for AIX, OS/2 and DOS Windows operating systems.

11.6.1 Threads on AIX Version 4

The AIX Version 4.1 kernel now supports multiple threads of control within a single process. Its libpthreads.a library has been written to the POSIX 1003.4a Draft 7 specification and is a linkable user library that provides user space threads service to an application.

There is also a set of kernel services provided to write kernel extensions for creating and managing threads as well as new locking services to assist the kernel developer working in the multiprocessor environment.

11.6.1.1 Multithreaded Programming

AIX 4.1 provides a thread-safe, reentrant version of the AIX C library, ***libc_r.a***, which allows multiple threads to execute concurrently. Some routines in the library do not have any change, but other routines required interface changes in the form of additional parameters to provide reentrancy. These routines have new names in the form of the original routine name plus an ***_r*** suffix. The C library has not been made fully reentrant, some SUN ONC RPC and NIS routines may not be reentrant.

11.6.1.2 Compatibility

AIX 4.1 offers the DCE Pthreads compatibility library, which is an implementation of the user threads on top of its kernel threads. It is based on the POSIX 1003.4a Draft 4 documentation with POSIX 1003.4a Draft 7 signal semantics, which is not binary compatible with AIX 3.2.5. However, the DCE pthread compatibility library exceptions are source level compatible with AIX 3.2.5 pthreads and the application must be recompiled. This compatibility library is available with the DCE LPP for backward compatibility.

Although, at present, all the POSIX threads documentation is on draft level, it seems that the POSIX 1003.4a Draft 10 will become the POSIX standard. Therefore, we strongly suggest to create applications that use the AIX Pthreads based on POSIX 1003.4a Draft 7, which is closer to the soon-to-be POSIX standard.

To compile a program using AIX 4.1 BOS pthreads, we need to use the `cc_r` or `xlc_r` compiler, the pthread.h header file, and link the program with the libpthreads.a library. With the threads compatibility packet, DCE for AIX provides a `cc_r4`/`xlc_r4` compiler (compatible to `cc_r` or `xlc_r` in AIX 3.2.5), the pthread_exc.h header (to use the exception handling mechanism) and the libdcelibc_r.a and libdcepthreads.a libraries (compatible to libc_r.a and libpthreads.a libraries in AIX 3.2.5).

Note: In AIX Version 4.1, all ***signal handlers*** are installed on a per-process basis. Hence, programs installing signal handlers need to be aware that they may replace existing handlers installed by other threads. The DCE Pthreads compatibility library maintains the signal semantics of POSIX 1003.4a Draft 7, which is different from that of Draft 4 (see 11.5.1, “Signals” on page 227). Even if your application compiles without any errors, the pthreads signal behavior may be incorrect.

11.6.1.3 AIX dbx Debugger

An application program can be debugged with the `dbx` command. The `dbx` is not part of DCE; it is an AIX operating-system tool. And it is fully thread-aware. It allows you to:

- Display and switch the current thread
- Display general information on alert, task control block and stack of a particular thread
- Display mutexes, condition variables and attribute objects

11.6.1.4 Fork

The POSIX 1003.4a Draft 7 specification defines a new function, `forkall()`, that solves the problem of inconsistent data between the child and the parent by requiring that the child contain replicas of all threads of the parent. However, this routine is not implemented on AIX Version 4.1. It would solve the problem of cancelling threads without notice, but introduces a new one, which is how to deal with threads suspended in system calls.

11.6.1.5 Non-Blocking I/O Support

In AIX Version 4.1, all the I/O calls are supported to block only the calling thread and not the entire process.

11.6.1.6 Scheduling

The DCE scheduling policies, SCHED_FG_NP and SCHED_BG_NP, are mapped onto SCHED_OTHER in AIX. Also, the maximum and minimum scheduling priorities for these policies are mapped to the default values.

11.6.2 Threads on AIX Version 3.2.5

This version of AIX does not support kernel threads. The user-level pthreads library is based on the POSIX 1003.4a Draft 4 and basically has all the features or suffers from all the restrictions described in 11.5, “More Advanced Threads Topics in UNIX” on page 227.

The dbx (see 11.6.1.3, “AIX dbx Debugger” on page 230), the reentrant libc_r.a jacket library and the cc_r/xlc_r compilers are also available on AIX Version 3.2.5.

The DCE Threads for AIX Version 3.2.5 are available as part of the DCE Base Services and as a separate licensed program product. AIX DCE Threads/6000 is available for users that need threads capability without DCE.

11.6.3 Threads on OS/2 Warp

For OS/2 users, threads are well known since OS/2 has had multitasking and threads support built-in since its first days. OS/2′s DCE implementation is a mapping of the DCE Threads API (based on POSIX 1003.4a Draft 4) to the corresponding OS/2 APIs (non POSIX).

DCE Threads APIs and OS/2 threads APIs can coexist within the same process, but they cannot interoperate. That is, you can use both APIs within the same process, or even within the same thread, but you cannot, for example, use one set of APIs to wake up a thread that is blocked in the other set of APIs.

From a performance perspective, DCE threads APIs are a layer on top of OS/2 threads APIs, and are therefore (insignificantly) slower. So, if portability is an issue, DCE threads are preferred over the native threads.

11.6.4 Threads for DOS Windows

The current IBM DCE for DOS Windows, based on OSF DCE 1.0.2 and the threads programming environment, is provided by the IBM DCE Client SDK. The use of the pthreads programming interface requires some precautions in the following areas:

- ***Stack segments and compiler behavior*** — Each DCE thread runs on its own stack which is allocated in a separate segment. This means that the stack segment (SS) is not equal to the data segment (DS). So, the standard assumption made by most DOS C compilers that SS equals DS does not hold true. The code has to be compiled with the /Aw flag causing a C compiler to generate a warning if it sees a code that assumes SS is equal to DS. Typically, this is where the stack pointers are passed to APIs.
- ***Non-preemptive Threads*** — Under IBM DCE for DOS Windows, threads are non-preemptive. The multitasking behavior of DOS Windows applications is achieved by the cooperative processing of messages by DOS Windows tasks, and not by preemption or interrupts. However, the OSF DCE implementation of pthreads uses a preemptive timer to schedule threads. Since DOS Windows timers are non-preemptive (they just post message), timers cannot be relied upon to schedule threads. Therefore, threads are reliably

scheduled only by using explicitly yielding calls, such as pthread_yield(), pthread_cond_wait() and DCEyield().

- ***No reentrant libraries needed*** — To guarantee that the threads are scheduled, you can set a timer and call pthread_yield() whenever you receive a timer message.

  Since a thread cannot be preempted asynchronously, a thread always yields synchronously, and when it yields, it is always inside DCE and never inside any application libraries. Therefore, there is no possibility of re-entering a library and standard libraries do not need to be thread-safe or reentrant. The wrapper routines are not necessary on this platform. However, libraries in IBM DCE for DOS Windows must be thread-safe because they use threads. This is true for any user program or library that uses threads. The synchronization is the same as other system environments (use of condition variable APIs and mutex APIs).

- ***Yielding to DOS Windows*** — Because of the non-preemptive nature of the DOS Windows environment, DCE applications have to be compatible with DOS Windows in order to allow other applications to run. In order to be compatible with DOS Windows, a DCE application must process messages and yield control to DOS Windows. The threads subsystem handles this by calling its *blocking API*. It yields the processor if no thread is ready to run. However, long-running threads that never block must call a DCEyield() to give other DOS Windows applications a chance to run.

- ***Stack size limitation*** — Since the DOS program stack must fit within a segment and the DOS Windows stack segment has a 16-byte overhead, the thread stack size is limited to 64 K, minus 16 bytes.

- ***Default stack size*** — Since DOS Windows must allocate real memory for stacks, the DCE default stack size has been reduced from 30 K to 12 K. The default stack size can be modified by the application. Certain applications ported from UNIX that have large numbers or sizes of arguments defined in their IDL files may have to increase the threads′ stack size.

- ***Pthread callback APIs*** — The pthread_create(), pthread_once() and pthread_keycreate() calls require you to pass to a callback API within your application. At the point in the program where one of these APIs is called, the value of the DS register must be equal to the data segment assumed by the callback API. This is normally the default data segment, unless you have to change DS explicitly. If the callback API assumes a different data segment, the pointer to the callback API should be the address returned by the MakeProcInstance() call.

Appendix A. DCE Application Examples

The following chapter will give you a starting point and a reference for DCE application examples. For further information about DCE application examples and DCE developing, consult the redbook *Developing DCE Applications for AIX, OS/2 and Windows* or one of the DCE Application Developing Guide manuals.

If you install the AIX fileset `dce.tools`, you will receive some DCE application examples. The DCE examples are located in the */usr/lpp/dce/examples* directory. Following is a list of the examples with a short description of their purposes:

- ***Time Provider*** (/usr/lpp/dce/examples/dts) — This example contains samples for time providers. Four sample external time-provider interface programs are provided.
- ***Timop*** (/usr/lpp/dce/examples/timop) — The example timop (Time Operations Sample Application) is a tutorial DCE application example. It exercises the basic DCE technologies: threads, RPC, security, directory, and time.
- ***ACL manager*** (/usr/lpp/dce/examples/acl_mgr) — The ACL manager example is a client/server application that demonstrates how one would go about writing an ACL manager. It is not meant to be an efficient use of storage, nor is it a complete manager. It′s sole purpose is to instruct one on how to start the task of writing an ACL manager.
- ***Bank demo program*** (/usr/lpp/dce/examples/bank) — The DCE Motif Bank demo is a client/server application that exploits all core services of DCE (threads, RPC, CDS, Security and DTS). An OSF/Motif front-end provides a graphical user interface to operations of the bank.
- ***RPC client/server examples*** (/usr/lpp/dce/examples/type_mgr) — This program implements a simple client/server distributed application, along with a management (administration) application. The actual application RPC operations implemented are trivial; the intention of the example is to demonstrate particular techniques that can be abstracted to production applications.
- ***Documentation examples*** (/usr/lpp/dce/examples/pubs) — This are four samples, which are described in the DCE Application Development Guide.
- ***Greet*** (/usr/lpp/dce/examples/pubs/greet) — One of the popular sample, the `greet` example, is in this directory.
- ***Demo*** (/usr/lpp/dce/examples/demo) — This directory contains the source code for a generic sample for a DCE client/server application.
- ***SVC*** (/usr/lpp/dce/examples/svc) — This directory comprises two new samples, which use the new DCE Serviceability API. They were developed mainly during the writing of the OSF DCE Application Development Guide chapter on Serviceability.

The examples comprise a *Steps* or a *README* file, which contains instructions for compilation and execution. These examples use five different types of source files to create, compile and run the DCE RPC examples:

- ***.idl***: — This file contains declarations for the RPC interface that the client and the server will be sharing. The .idl file is used later by the IDL compiler to create the header and stub files.
- ***.acf***: — This file contains information that changes how an IDL compiler interprets the interface definition (.idl file). The definition of an .acf file is optional.
- ***server.c***: — This file contains the RPC initialization calls needed to accept, control and terminate a communication with client applications.
- ***<manager>.c***: — This file contains the actual code that defines the services provided by the server program.
- ***<client>.c***: — This file contains the Remote Procedure Calls needed to contact the server application and the logic to consume the services provided by the server application.
- ***Makefile***: — This file is used to create the executable object code.

List of Abbreviations

ACL	access control list
ANSI	Amercian National Standards Institute
ATM	asynchronous transfer mode
CDMF	Common Data Masking Facility
CDS	Cell Directory Service
CICS	Customer Information Control System
CMA	Concert Multithread Architecture
CMIP	common management interface protocol
CMVC	Configuration Management and Version Control
CORBA	common object request broker
COSE	Common Open Software Environment
DAP	directory access protocol
DCE	Distributed Computing Environment
DES	data encryption standard
DFS	Distributed File System
DNS	domain name service
DSOM	distributed system object model
DSS	distributed system services
EPAC	extended privilege attribute certificate
ERA	extended registry attributes
FCS	fibre channel standard
FLDB	Fileset Location Database
GDA	Global Directory Agent
GDS	Global Directory Service
HACMP	High Availability Cluster Multi-Processing
IBM	International Business Machines Corporation
ICC	Initial Container Creation (ACL)
IDL	interface definition language
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IOC	Initial Object Creation (ACL)
IP	internet protocol
ISO	International Standardization Organization
ITSO	International Technical Support Organization
LAN	local area network
LFS	Local File System
LSE	LAN Server Enterprise
LRPC	local RPC
MAN	metropolitan area network
MQI	message queuing interface
MTPN	Multiprotocol Network Transport
NCACN	Network Computing Architecture Connection Based Protocol
NCADG	Network Computing Architecture Datagram Protocol
NFS	Network File System
NIS	Network Information System
NSI	Name Service Interface
NTP	network time protocol
OLTP	on-line transaction processing
OMG	Object Management Group
ONC	Open Network Computing
OSF	Open Software Foundation
PAC	privilege attribute certificate
PGO	principal, group, organization
RAID	redundant array of independent disks
RDBMS	relational database management system
RPC	remote procedure call
SCM	System Control Machine
SLC	Secured Logon Coordinator (NetSP)
SNG	Secured Network Gateway (NetSP)
SNMP	simple network management protocol

SOM	system object model	***UDP***	user datagram protocol
SQL	structured query language	***UUID***	universal unique identifier
TCP	transmission control protocol	***WAN***	wide area network
TPI	time provider interface	***XMP***	X/Open management protocol

Index

ITSO Technical Bulletin Evaluation RED000

International Technical Support Organization
Understanding OSF DCE 1.1 for AIX and OS/2
October 1995

Publication No. SG24-4616-00

Your feedback is very important to help us maintain the quality of ITSO Bulletins. **Please fill out this questionnaire and return it using one of the following methods:**

- Mail it to the address on the back (postage paid in U.S. only)
- Give it to an IBM marketing representative for mailing
- Fax it to: Your International Access Code + 1 914 432 8246
- Send a note to REDBOOK@VNET.IBM.COM

Please rate on a scale of 1 to 5 the subjects below.
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction ____

Organization of the book	____	Grammar/punctuation/spelling	____
Accuracy of the information	____	Ease of reading and understanding	____
Relevance of the information	____	Ease of finding information	____
Completeness of the information	____	Level of technical detail	____
Value of illustrations	____	Print quality	____

Please answer the following questions:

a) If you are an employee of IBM or its subsidiaries:

Do you provide billable services for 20% or more of your time? Yes____ No____

Are you in a Services Organization? Yes____ No____

b) Are you working in the USA? Yes____ No____

c) Was the Bulletin published in time for your needs? Yes____ No____

d) Did this Bulletin meet your needs? Yes____ No____

If no, please explain:

____

____

What other topics would you like to see in this Bulletin?

____

____

What other Technical Bulletins would you like to see published?

____

Comments/Suggestions: **(THANK YOU FOR YOUR FEEDBACK!)**

_______________________ _______________________
Name Address

_______________________ _______________________
Company or Organization

_______________________ _______________________
Phone No.

ITSO Technical Bulletin Evaluation **RED000**

SG24-4616-00

Cut or Fold Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM International Technical Support Organization
Department JN9, Building 821
Internal Zip 2834
11400 BURNET ROAD
AUSTIN TX
USA 78758-3493

Fold and Tape **Please do not staple** Fold and Tape

SG24-4616-00

Cut or Fold Along Line

Printed in U.S.A.

SG24-4616-00