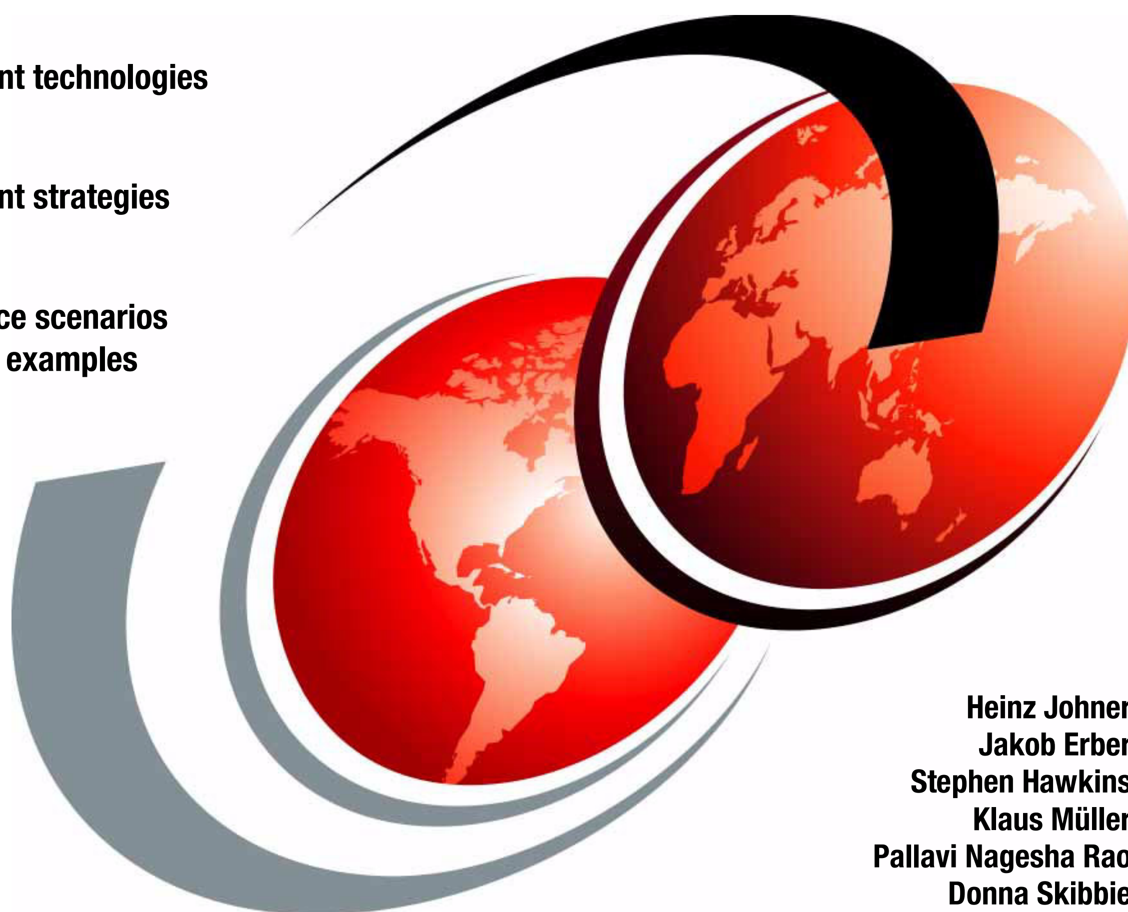
IBM

DCE Replacement Strategies

Replacement technologies

Replacement strategies

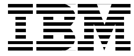
Best practice scenarios and coding examples



Heinz Johner
Jakob Erber
Stephen Hawkins
Klaus Müller
Pallavi Nagesha Rao
Donna Skibbie

Redbooks

ibm.com/redbooks



International Technical Support Organization

DCE Replacement Strategies

June 2003

SG24-6935-00

Note: Before using this information and the products described, read the information in "Notices" on page xvii.

First Edition (June 2003)

This edition applies to Version 3.2 of IBM DCE, Version 1.3 of IBM Network Authentication Service, Version 3.9 of IBM Tivoli Access Manager, Version 3.2.2 of IBM Directory Server and to related products.

© Copyright International Business Machines Corporation 2003. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Tables xiii

Figures xv

Notices xvii
Trademarks xviii

Preface xix
The team that wrote this redbook xx
Become a published author xxi
Comments welcome xxii

Part 1. Description of the DCE replacement strategies 1

Chapter 1. DCE review 3
1.1 Defining DCE 4
1.2 Who uses DCE 4
1.3 What DCE does 5
1.3.1 Threads 5
1.3.2 RPC 5
1.3.3 Security core 6
1.3.4 GSS-API 9
1.3.5 Directory 9
1.3.6 Time 9
1.3.7 Cross component 9
1.4 The DCE environment 11
1.5 Application dependencies on DCE 12
1.5.1 Direct dependencies 12
1.5.2 Indirect dependencies 13
1.5.3 No dependencies 15
1.6 Summary of DCE review 15

Chapter 2. Replacement technologies 19
2.1 Criteria for selecting the technologies 20
2.1.1 Compliance with industry standards 20
2.1.2 Coverage of predominant DCE services 20
2.1.3 Ease of migration 20
2.1.4 Similarity to DCE services 20
2.1.5 Technologies considered strategic 21

© Copyright IBM Corp. 2003. All rights reserved.

2.1.6 Support of predominant platforms 21
2.1.7 Support of predominant programming languages 21
2.1.8 Availability of IBM implementations 21
2.2 Technologies for C/C++ applications 21
2.2.1 aznAPI 22
2.2.2 CORBA 23
2.2.3 DCE RPC 26
2.2.4 DCE UUID 26
2.2.5 Kerberos 26
2.2.6 LDAP 28
2.2.7 Network Time Protocol 30
2.2.8 Platform auditing 31
2.2.9 Platform logging and messaging 31
2.2.10 POSIX 1003.1c threads 32
2.2.11 Web services 33
2.3 Technologies for Java applications 34
2.3.1 J2EE application environment 34
2.3.2 Standards for the J2EE 35
2.3.3 DCE services that can be replaced by J2EE 37
2.3.4 IBM implementation of J2EE: WebSphere Application Server 39
2.3.5 Additional information on IBM WebSphere Application Server 40
2.4 Summary 47

Chapter 3. Replacement strategies 51
3.1 Replacement strategies for C/C++ applications 52
3.1.1 Auditing 52
3.1.2 Authentication 53
3.1.3 Authorization, PAC, and UUID 53
3.1.4 Backing store 56
3.1.5 Configuration 56
3.1.6 Delegation, GSS-API, and login 56
3.1.7 Directory 58
3.1.8 Extended Registry Attributes 58
3.1.9 Event management 59
3.1.10 GSS-API 59
3.1.11 Host management 59
3.1.12 Integrated login 59
3.1.13 Login 60
3.1.14 Messaging 60
3.1.15 PAC 61
3.1.16 Password strength 61
3.1.17 Protection 62
3.1.18 Registry 62

3.1.19 RPC services . . . 63
3.1.20 Serviceability . . . 66
3.1.21 Threads . . . 67
3.1.22 Time . . . 69
3.1.23 UUID . . . 69
3.2 Replacement strategy for Java applications . . . 69
3.2.1 Determining a new architecture . . . 70
3.2.2 Revising the application environment for the new architecture . . . 72
3.2.3 Rewriting the DCE applications to the new architecture . . . 73
3.3 Replacement strategies for mixed applications . . . 73
3.3.1 CORBA interoperability . . . 73
3.3.2 Java Native Interface . . . 73
3.3.3 JCA and JNI . . . 74

Chapter 4. Using DCE data with IBM Network Authentication Service . . 75
4.1 Introduction . . . 76
4.2 Migrating DCE data to an LDAP directory . . . 76
4.3 Configuring IBM Network Authentication Service . . . 76
4.4 Managing the data in a shared environment . . . 77
4.5 Removing DCE-specific data . . . 79
4.6 Details about shared data . . . 79
4.7 Details about non-shared data . . . 81

Chapter 5. Using DCE objects with IBM Tivoli Access Manager . . . 85
5.1 Introduction . . . 86
5.2 Data representation . . . 86
5.3 Configuration scenarios . . . 92
5.3.1 Scenario 1 . . . 92
5.3.2 Scenario 2 . . . 92
5.3.3 Scenario 3 . . . 94
5.4 Managing objects in a shared environment . . . 96
5.4.1 Creating a user with IBM Tivoli Access Manager . . . 96
5.4.2 Creating a group with IBM Tivoli Access Manager . . . 97
5.4.3 Adding a member to a group using IBM Tivoli Access Manager . . . 97
5.4.4 Deleting a user using IBM Tivoli Access Manager . . . 98
5.4.5 Deleting a group using IBM Tivoli Access Manager . . . 98
5.4.6 Removing a member from an IBM Tivoli Access Manager group . . . 99
5.4.7 Creating a principal with DCE . . . 99
5.4.8 Creating a DCE group . . . 100
5.4.9 Adding a member to a group using DCE . . . 100
5.4.10 Deleting a user using DCE . . . 101
5.4.11 Deleting a group using DCE commands . . . 101
5.4.12 Removing a member from a group with DCE commands . . . 102

5.4.13 Sharing policies 102
5.4.14 Attaching a DCE policy 102
5.4.15 Deleting a shared DCE policy 103

Chapter 6. Binary structure of DCE ERA data in LDAP 105
6.1 Recap: The DCE to LDAP migration process 106
6.2 Reading binary DCE ERA data in LDAP 106

Part 2. Replacement sample scenarios 109

Chapter 7. Common replacement considerations 111
7.1 How to read the example scenarios 112
7.2 Common assumptions in the sample scenarios 112
7.3 Simplifications in the sample scenarios 112
7.4 Security considerations 113
7.5 Performance considerations 113
7.6 Using an LDAP directory 114
7.6.1 LDAP security considerations 114
7.6.2 Availability and performance considerations 115
7.7 SSL implementation hints 116
7.7.1 SSL and TLS overview 116
7.7.2 Uses of SSL 117
7.7.3 Using SSL in the replacement scenarios 117
7.7.4 IBM GSKit 118
7.7.5 Authentication with certificates 119
7.7.6 Using self-signed certificates 120
7.7.7 Using certificates from a Certificate Authority (CA) 121
7.7.8 Additional hints and considerations 123

Chapter 8. Scenario 1: GSS-API application 125
8.1 Scenario description 126
8.1.1 Initial application with DCE dependencies 126
8.1.2 Revised application without DCE dependencies 128
8.2 DCE application 129
8.2.1 Configuring and running the DCE application 130
8.2.2 Application client 130
8.2.3 Application server 134
8.3 Replacement roadmap 138
8.3.1 Software requirements 138
8.3.2 Migration of DCE security registry to IBM Directory Server 139
8.3.3 Configuring IBM Network Authentication Service 144
8.3.4 Configuring IBM Tivoli Access Manager 145
8.3.5 Configuring the Windows Kerberos client 149
8.3.6 Revising the application 151

8.3.7 Cleaning up the DCE related information in the IBM Directory 151
8.4 Revised application discussion 152
8.4.1 Configuring and running the revised application 152
8.4.2 Application client 153
8.4.3 Application server 155
8.5 Administration considerations and interfaces 157
8.5.1 Administration during the migration process 157
8.5.2 Administration after the migration process 164
8.5.3 IBM Network Authentication Service administration interface. 165
8.5.4 IBM Tivoli Access Manager administration interface 167
8.6 Discussion and conclusions 168

Chapter 9. Scenario 2: Non-secure RPC application 171
9.1 Scenario description 172
9.1.1 Initial application with DCE dependencies 172
9.1.2 Revised application without DCE dependencies. 174
9.2 DCE application. 177
9.2.1 Configuring and running the DCE application 177
9.2.2 Application client 178
9.2.3 Application server 178
9.3 Replacement roadmap 180
9.3.1 Software requirements 180
9.3.2 Installing and configuring WebSphere Application Server 180
9.3.3 Revising the application 180
9.3.4 Removing DCE 181
9.4 Revised application discussion 181
9.4.1 Building, configuring, and running the revised application 181
9.4.2 CORBA IDL file 183
9.4.3 Application client 184
9.4.4 Application server 189
9.5 Administration considerations and interfaces 192
9.6 Discussion and conclusions 192

Chapter 10. Scenario 3: Secure RPC application #1 195
10.1 Scenario description 196
10.1.1 Initial application with DCE dependencies 196
10.1.2 Revised application without DCE dependencies. 198
10.2 DCE application. 203
10.2.1 Configuring and running the DCE application 203
10.2.2 Application interface definition 204
10.2.3 Application client 205
10.2.4 Application server 207
10.3 Replacement roadmap 210

10.3.1 Software requirements 211
10.3.2 Installing and configuring IBM WebSphere Application Server . . . 211
10.3.3 Configuring WebSphere Application Server security 212
10.3.4 Configuring the application client 222
10.3.5 Developing the application 223
10.3.6 Configuring IBM Directory Server 223
10.3.7 Assembling the scenario application 223
10.3.8 Deploying and starting the application 227
10.3.9 Running the application client 229
10.4 Revised application discussion 230
10.4.1 Enterprise bean wrappers 230
10.4.2 CORBA IDL file 231
10.4.3 Application client 231
10.4.4 Application server 236
10.4.5 Java Native Interface 238
10.4.6 J2EE Connector Architecture 241
10.5 Administration considerations and interfaces 245
10.6 Discussion and conclusions 246

Chapter 11. Scenario 4: Secure RPC application #2 247
11.1 Scenario description 248
11.1.1 Initial application with DCE dependencies 248
11.1.2 Revised application without DCE dependencies 248
11.2 DCE application 251
11.3 Replacement roadmap 251
11.3.1 Software requirements 252
11.3.2 Installing and configuring IBM WebSphere Application Server . . . 252
11.3.3 Configuring WebSphere Application Server security 252
11.3.4 Configuring the application client 253
11.3.5 Developing the application 254
11.3.6 Configuring IBM Directory Server 255
11.3.7 Assembling the scenario application 255
11.3.8 Deploying and starting the application 258
11.3.9 Running the application client 259
11.4 Revised application discussion 259
11.4.1 Application client 260
11.4.2 Application server 260
11.5 Administration considerations and interfaces 262
11.6 Discussion and conclusions 262

Part 3. Appendixes 265

Appendix A. Scenario 1: Source code listings 267
Application with DCE dependencies 268

Makefile for application client 268
Makefile for application server 269
DCE dependent application client 270
DCE dependent application server 276
Authorization module with DCE dependencies 281
Utility source of the DCE dependent application 286
Header file for utility source 291
Revised application without DCE dependencies 292
Makefile for application client 293
Makefile for application server 294
Revised application client 294
Revised application server 299
Authorization module using aznAPI 304
Utility source of the revised application 309
Header file for utility source 317
Application configuration file 319

Appendix B. Scenario 2: Source code listings 331
Application with DCE dependencies 332
Makefile for the AIX platform 332
Makefile for the Windows platform 334
IDL source 335
DCE dependent application client 335
DCE dependent application server 337
Application server logic 341
Revised application without DCE dependencies 342
Makefile for the AIX platform 343
Makefile for the Windows platform 344
CORBA IDL file 346
Header file for C wrapper functions 346
C++ client for the revised application 346
C client for the revised application 350
Header file for CORBA servant 350
Servant implemention 352
Revised application server 353
Revised application logic 360
Properties file for client 361
Properties file for server 361

Appendix C. Scenario 3: Source code listings 363
Application with DCE dependencies 364
Makefile for the AIX platform 364
Makefile for the Windows platform 366

IDL source . . . 367
DCE dependent application client . . . 368
DCE dependent application server . . . 371
Application server manager . . . 377
Application server logic header . . . 380
Application server logic . . . 380
Common error handling . . . 382
Revised application without DCE dependencies . . . 382
JNI connection . . . 383
JNI connection factory class . . . 385
JNI connection manager . . . 386
JNI managed connection . . . 386
JNI connection meta data . . . 388
JNI managed connection factory interface . . . 388
JNI managed connection meta data . . . 390
JNI resource adapter meta data . . . 390
Enterprise bean wrapper . . . 391
Enterprise bean remote interface . . . 393
Enterprise bean home interface . . . 393
Deployment descriptor for the application . . . 393
Deployment descriptor for the enterprise bean . . . 394
Deployment descriptor for the resource adapter . . . 396
"C" application server . . . 396
JNI wrapper for the application server . . . 398
Header file for application server . . . 399
CORBA C++ client . . . 399
Properties file for CORBA C++ client . . . 402

Appendix D. Scenario 4: Source code listings . . . 403
Application with DCE dependencies . . . 404
Revised application without DCE dependencies . . . 404
Build script to create class and jar files . . . 404
Java client program . . . 405
Java stateless session bean . . . 406
Java EJB home interface . . . 408
Java EJB remote interface . . . 408
Java application exception . . . 409
EJB deployment descriptor . . . 409
Application client deployment descriptor . . . 410
Application deployment descriptor . . . 411
Application client security properties . . . 411

Appendix E. Additional material . . . 417

Locating the Web material 417
Using the Web material 418
System requirements for downloading the Web material 418
How to use the Web material 418

Abbreviations and acronyms 419

Related publications 421
IBM Redbooks 421
Other publications 421
Online resources 422
How to get IBM Redbooks 423

Index 425

Tables

1-1 DCE Server Information 11
1-2 Indirect dependencies for many DCE applications on DCE services . . 13
1-3 Indirect dependency for some DCE applications on DCE services 14
1-4 The dependencies of services 15
2-1 Summary of recommended replacement technologies 48
3-1 DCE server replacements for auditing 52
4-1 Shared data 79
4-2 Data specific to DCE 82
4-3 Data specific to IBM Network Authentication Service 82
4-4 Data stored in different attributes 83
10-1 Security policy 197

© Copyright IBM Corp. 2003. All rights reserved.

Figures

1-1 Relation of DCE to the application and platform 4
2-1 Authentication protocols. 40
2-2 J2EE connection process using RMI over IIOP 43
2-3 IBM WebSphere Application Server architectural overview. 45
2-4 Naming topology 46
3-1 Java client and back-end enterprise beans 70
3-2 Web clients, servlets, and back-end enterprise beans 71
3-3 JCA and JNI. 74
5-1 Sample DCE principal called user1 in LDAP 87
5-2 Sample IBM Tivoli Access Manager user called user1 in LDAP 88
5-3 Sample shared account called user1 89
5-4 Sample DCE group called group1 in LDAP 90
5-5 Sample IBM Tivoli Access Manager group called group1 in LDAP 91
5-6 Sample shared group called group1 91
7-1 The components of IBM GSKit (simplified) 118
7-2 Using self-signed certificates 120
7-3 Using certificates from a Certificate Authority 122
8-1 Authentication and authorization with DCE dependencies 128
8-2 Authentication and authorization without DCE dependencies 129
8-3 Migration environment with DCE dependencies 158
8-4 Replication and data access in a mixed environment 160
9-1 DCE RPC connection process using CDS. 173
9-2 CORBA connection process, using CORBA CosNaming service 175
10-1 Application with DCE dependencies 197
10-2 The revised application 199
10-3 Replacement roadmap. 202
10-4 SSL settings for IBM WebSphere Application Server 215
10-5 LDAP settings for IBM WebSphere Application Server 217
10-6 Name matching between client certificate and directory entry. 219
10-7 Global Security settings for IBM WebSphere Application Server 221
11-1 The revised application in the J2EE environment 249
11-2 Traditional and J2EE application layering 262

© Copyright IBM Corp. 2003. All rights reserved.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

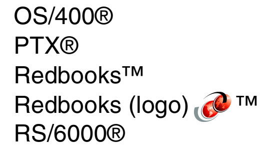
This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

© Copyright IBM Corp. 2003. All rights reserved.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®	OS/400®	SecureWay®
DB2®	PTX®	Tivoli®
DFS™	Redbooks™	TXSeries™
IBM®	Redbooks (logo)  ™	WebSphere®
OS/390®	RS/6000®	z/OS™

The following terms are trademarks of other companies:

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.

Preface

Since its inception in the early 1990s, the Distributed Computing Environment (DCE) has been used successfully in distributed, networked environments where a high level of security, distributed and heterogeneous platforms, and large numbers of systems, users, and groups were to be managed in a consistent way. DCE has been used in small and large organizations. Such organizations may be faced with other technologies that they consider strategic on which to build their future infrastructure and application framework.

This IBM® Redbook lists and recommends strategies that can be used to replace the DCE dependencies in such environments and to move to new technologies and products. It covers the following topics:

- DCE overview and recap
- Replacement technologies and strategies
- Replacement scenarios with sample application code
- Additional information

While this book covers replacement strategies for DCE, it does not cover replacement strategies for IBM products that use DCE, such as DFS™ and TXSeries™.

This book assumes that you are an executive, administrator, or developer of an IBM customer environment that uses IBM DCE for the AIX®, Solaris, and Windows platforms.

© Copyright IBM Corp. 2003. All rights reserved.

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world, working at the International Technical Support Organization, Austin Center.

Heinz Johner is a Certified Consulting IT Specialist for Information Security at IBM Switzerland. His primary areas of expertise are security architecture design and reviews. As a previous lead author of various IBM Redbooks™ on DCE, LDAP and other, security-related areas, he led the team at International Technical Support Organization, Austin Center, for the development of this book.

Jakob Erber is a Senior Software Engineer with Paranor AG, a software company in Switzerland. Paranor AG specializes in the field of mission-critical banking applications. He holds a degree in Computer Science from the Munich University of applied sciences. His areas of expertise range from design and implementation of business applications to the field of middleware, including DCE, CORBA, and J2EE.

Stephen Hawkins is a Consulting IT Architect with the IBM Software Group Application Integration and Middleware WebSphere® Services team. He has been an architect and developer of middleware and infrastructure for the past 20 years. His areas of expertise include system interoperability, transaction processing, security, and IT operations.

Klaus Müller is a Security Architect at IBM Global Services, Strategic Outsourcing. He has six years of experience in the middleware technology field and holds a degree in Physics from the University of Dortmund. Before joining IBM, he worked as a DCE/DFS consultant in the banking industry. His areas of expertise include UNIX, RS/6000®, middleware, and security technologies.

Pallavi Nagesha Rao is a Software Engineer at IBM India. She has three years of experience in the field of Information Technology. She holds a degree in Computer Science from Bangalore University. Her area of expertise is designing and developing applications for middleware, including J2EE application servers and transaction processing systems such as TX-Series.

Donna Skibbie is a Distributed Systems Architect at IBM Austin. She was a lead architect in developing the DCE replacement strategies and worked with the Internet Engineering Task Force and The Open Group on the Kerberos and DCE LDAP schemas. She holds a Masters Degree in Computer Science and has 15 years of experience in the distributed technology field. She has many years of experience in technical writing.

Thanks to the following person for his significant contribution to this book:

Grant Alvis Software Project Leader, IBM Austin

Thanks to the following people for their help and contributions to this project:

Phil Adams	IBM Austin
Axel Buecker	International Technical Support Organization, Austin Center
Ching-Yun Chao	IBM Austin
Garry Child	IBM Austin
Kristen Clarke	IBM Austin
Alaine DeMyers	IBM Austin
Terry Dunkle	IBM Austin
Gary Gerchak	IBM Austin
Laura Giovannetti	IBM Austin
Kevin Kelle	IBM Austin
Ut Le	IBM Austin
Robin Redden	IBM Austin
Walter Schmid	Paranor AG
Cindy Schneider	IBM Austin
Dick Sikkema	IBM Austin
Dawn Stokes	IBM Austin
Betsy Thaggard	International Technical Support Organization, Austin Center
Mary Trumble	IBM Austin
Leo Uzcategui	IBM Austin
Bill Wentworth	IBM Austin

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. JN9B Building 003 Internal Zip 2834
  11400 Burnet Road
  Austin, Texas 78758-3493

Description of the DCE replacement strategies

This part provides a description of the DCE replacement strategies.

© Copyright IBM Corp. 2003. All rights reserved.

1

DCE review

This chapter presents a brief review of DCE. The purpose of this introductory chapter is to establish a common way of thinking about each DCE service and how your environment uses each service. The remainder of this book uses this common knowledge when it describes the strategies that you can use to replace each DCE service. For a detailed overview of DCE, the following book and document are recommended:

- *IBM DCE Version 3.2 for AIX and Solaris: Introduction to DCE*
- *DCE Overview*, The Open Group (`http://www.opengroup.org`)

© Copyright IBM Corp. 2003. All rights reserved.

1.1 Defining DCE

DCE is middleware software consisting of development tools, servers, and commands. Customers use it to create, run, and manage distributed client and server applications (called DCE applications) in a heterogeneous computing environment. The specifications for DCE were developed by a consortium of software vendors under the organization of The Open Group (formerly the Open Software Foundation). Many software vendors, including IBM, have developed implementations of DCE on many different platforms.

Architecturally, DCE acts as a layer of software between the platform and network on the one hand, and the DCE application on the other. DCE enables the DCE applications to interact with a group of heterogeneous platforms as if they were a single system (known as a DCE cell). Figure 1-1 illustrates DCE in relation to the application and platform.

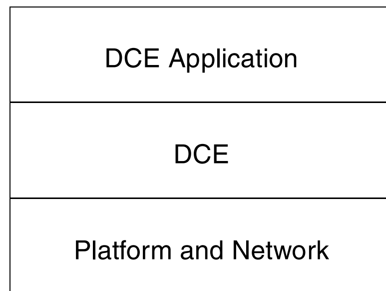


Figure 1-1 Relation of DCE to the application and platform

1.2 Who uses DCE

DCE has three groups of users: developers, administrators, and, to a lesser extent, end users. Developers use the DCE development tools to create DCE applications.

Some popular DCE applications are:

- Distributed File System (DFS)
- IBM TXSeries
- High Performance File System (HPFS)
- IBM Tivoli® Access Manager (formerly named Tivoli Policy Director)
- AIX Parallel System Support Program (PSSP)

Note: Some DCE specifications state that DFS is a component of DCE, while other DCE specifications state that DFS is a user of DCE. In this paper, DFS is considered to be a user of DCE — in other words, a DCE application.

Administrators use the DCE servers and DCE commands to manage the environment for running DCE applications. End users sometimes run DCE commands, such as the DCE login command.

1.3 What DCE does

DCE provides a set of services that a DCE application can use. These services are grouped into the following components, further explained in the sections that follow:

- Threads
- Remote procedure call (RPC)
- Security core
- Generic security services application programming interface (GSS-API)
- Directory services
- Global Directory Agent and Intercell
- Time synchronization
- Cross component

1.3.1 Threads

The threads component contains a single service: threads. This service provides a user-level threads compatibility library, which developers can use to create a multi-threaded DCE application that is independent of the threads model of the underlying platform. The only users of this service are developers.

1.3.2 RPC

The RPC component contains a group of services that a developer can use to create a DCE application. Application clients call procedures that are handled by remote application servers. Using RPC, clients can call remote procedures similar to the way that they call local procedures. The RPC services are:

- RPC basics
- RPC endpoint
- RPC namespace
- RPC security

RPC basics

The RPC basics service provides the basic mechanism for RPC. This includes defining remote interfaces using an Interface Definition Language (IDL), serializing the RPC call so it can be transmitted over the communications line, transmitting the call over a communications line using either a connection-oriented or connectionless communications protocol, and converting the data in the RPC call to the format of the native platform (if required). The only users of the RPC basics service are developers.

RPC endpoint

The RPC endpoint service provides a way for a client to locate the port to use in communication with a server. The primary users of this service are developers. Other users of this service are administrators who sometimes use DCE commands to clean up RPC endpoint information.

RPC namespace

The RPC namespace service provides a way for a client to locate the host name and binding information of a server that offers the desired procedure. The primary users of this service are developers. Other users of this service are administrators who configure RPC namespace information in the directory. (See section 1.3.5, “Directory” on page 9.)

RPC security

The RPC security service provides a way for developers to invoke the DCE authentication service, the DCE protection service, or both in an RPC call. The DCE authentication and protection services are described in section 1.3.3, “Security core” on page 6. By invoking the authentication service in an RPC call, the client must prove its identity to the server and the server must prove its identity to the client. By invoking the protection service in the call, the data transmitted in the call is protected through encryption, integrity, or both. The only users of the RPC security service are developers.

1.3.3 Security core

The security core component of DCE provides services for securing a DCE application. These services are:

- Authentication
- Authorization
- Delegation
- Login
- Privilege attribute certificate (PAC)
- Protection

- Registry
- Other

Authentication

The authentication service provides a way for clients and servers to pass authentication information to each other to prove their identities. This service has no direct users because it can be accessed by means of the GSS-API (see section 1.3.4, “GSS-API” on page 9) and RPC security services only.

Authorization

The authorization service provides functions that a server can use to control access to its services. The server does this by:

1. Comparing the identity or PAC of a client with the permissions configured for the service
2. Using the comparison to determine whether the client is authorized to use the service

The authorization service also provides functions that an administrator can use to configure permissions for services, usually in access control lists (ACLs). The primary users of this service are administrators and end users who use DCE commands to configure ACLs. Developers also use this service programmatically both to configure ACLs and to add access control functionality to an application server.

Delegation

The delegation service provides a way for servers to assume the identities of clients. The only users of this service are developers.

Login

The login service provides a way for a client to obtain a credential from a trusted third party that authenticates the identity of the client. The primary users of the login service are end users, who access it via the DCE login command. Developers also use this service programmatically, and administrators also access the service via the DCE login command.

PAC

The PAC service provides a way for a trusted third party to map the identity of a client to a PAC and for a server to retrieve the contents of the PAC. The PAC contains the client ID and an ID of each group in which the client is a member. This information is used for authorization. The primary users of the PAC service are end users who access it by means of the DCE login command. Developers also use this service, programmatically, to retrieve the contents of the PAC.

Protection

The protection service provides a way to protect the privacy of information through encryption and to protect the integrity of information through checksums. This service has no direct users because it can be accessed only by means of the GSS-API (section 1.3.4, “GSS-API” on page 9) and RPC security services.

Registry

The registry service provides a way to configure a registry database. The registry database contains information about each user principal, the groups and organizations in which they are members, and the security policies that must be enforced. The primary users of the registry service are administrators who use this service to manage the DCE registry database. Other infrequent users of this service are developers who program a DCE application to manage the DCE registry database.

Other

The other security services are optional. They are:

Auditing	Auditing provides a way for a server to log events that might compromise the security system. The primary users of this service are administrators who configure and view the audit log. Other users of this service are developers who use this service to add the capability of auditing a user-written server.
Extended Registry Attribute	Extended Registry Attributes (ERAs) provide a way to extend the registry database with additional attributes. The primary users of this service are administrators who configure additional attributes in the registry database. Other users of this service are developers who program a DCE application to configure or retrieve the contents of additional attributes.
Integrated login	Integrated login provides a way to integrate a DCE login with an operating system login. The primary users of this service are end users. Other users of this service are administrators and developers.
Password strength	Password strength provides a way to validate a password against a set of rules. The passwords are validated during a password change request. The primary users of this service are administrators who define the strength checking

rules and end users who change their passwords. Other infrequent users of this service are developers who write customized password checking routines.

1.3.4 GSS-API

The GSS-API component contains a single service: GSS-API. This service provides a way for developers to invoke the DCE authentication service, the DCE protection service, or both through the standard programming interfaces of GSS-API. Note that the GSS-API is a standard that is defined outside DCE, but DCE, just like other security middleware software, supports the GSS-API. The DCE authentication and protection services are described in section 1.3.3, “Security core” on page 6. The only users of this service are developers.

1.3.5 Directory

The directory component contains a single service: directory (also known as the *Cell Directory Service*, or CDS). This service, used internally by other DCE components, provides a DCE application with a common repository for storing information. The directory could be located in the same DCE cell or, through the use of the *Global Directory Agent* (GDA), in a different DCE cell. The primary users of this service are administrators. Other users of this service are developers.

1.3.6 Time

The time component contains a single service: time. This service (also known as the *Distributed Time Service*, or DTS) synchronizes the time among all of the nodes in the DCE cell. The primary users of this service are administrators. Other infrequent users of this service are developers.

1.3.7 Cross component

The cross component category contains a group of additional services that are useful in DCE applications. The services in this category are:

- Backing store
- Configuration
- Event management
- Host management
- Messaging
- Serviceability
- Universal Unique Identifier (UUID)

Backing store

The backing store service provides a way for an application server to create and use a backend database (for example, to store dynamic run-time information). The only users of this service are developers.

Configuration

The configuration service provides a way to configure DCE. The primary users of this service are administrators. Developers also use this service.

Event management

The event management service provides a way to manage events in a DCE application. The only users of this service are developers.

Host management

The host management service provides a way to manage data and processes on local and remote hosts. The primary users of this service are administrators who use DCE commands to manage remote hosts. Other users of this service are developers who manage remote hosts programmatically.

Messaging

The messaging service provides a way to display text associated with errors and system status in English and other written languages. The primary users of this service are developers, who use it programmatically to develop applications that display messages in English and, optionally, other languages. Developers also use it to retrieve text associated with DCE system errors.

Serviceability

The serviceability service provides a way to log or display informational, error, and debug messages. The users of this service are administrators and developers. Administrators specify the type and amount of information that is logged by DCE and the DCE applications. Then, administrators can view the logs that are generated. Developers use this service programmatically to include production and debug logging capability in their DCE applications and to obtain information from DCE logs.

UUID

The UUID service generates UUIDs. The ability to generate UUIDs is a requirement of the RPC basics and authorization services. The primary users of the UUID service are developers who programmatically configure UUIDs for RPC. Other users of this service are administrators who use commands to configure UUIDs for servers and, by means of the authorization service, for users and groups.

1.4 The DCE environment

Some DCE services require the use of DCE servers, and some of these DCE servers maintain DCE databases. The DCE servers and any DCE databases managed by these DCE servers must be configured on the network and managed by a DCE administrator. The DCE applications, DCE servers, and DCE databases are collectively referred to as a *DCE environment*. Table 1-1 lists each of the DCE servers and provides the following information about each server:

- the name of the DCE process that implements the DCE server
- the database (if any) that is maintained by the DCE server
- the DCE services that directly depend on the DCE server

Table 1-1 DCE Server Information

DCE server	DCE process	DCE database	DCE authentication service
DCE KDC server	secd	None (uses DCE registry service to access authentication data in the DCE registry database)	DCE authentication service
DCE directory server	cdsd	DCE namespace database	DCE directory service
DCE host server	dced	DCE endpoint database	DCE host management service
DCE privilege server	secd	None (uses the DCE registry service to access privilege data in the DCE registry database)	DCE PAC service
DCE registry service	secd	DCE registry database	DCE registry service
DCE time server	dtsd	none	DCE time service

Note that the documentation that comes with DCE often refers to the DCE authentication, privilege, and registry servers as one DCE server: a DCE security server. In this book, they are referred to as three independent DCE servers.

1.5 Application dependencies on DCE

For each DCE service, an application might have direct dependencies, indirect dependencies, a combination of direct and indirect dependencies, or no dependencies.

1.5.1 Direct dependencies

An application directly depends on a DCE service if it makes direct calls to the service. To remove this dependency, the application must be revised either to not use the service or to use a replacement for the service. For example, if an application makes direct calls to the DCE authorization service, the application has a direct dependency on this service. To remove this dependency, the application must be revised either to not use an authorization service (which is rather unlikely) or to use a replacement for the DCE authorization service.

Many DCE applications have direct dependencies on the following DCE services:

- Authorization
- GSS-API
- Login
- Messaging
- PAC
- RPC basics
- RPC endpoint
- RPC namespace
- RPC security
- Threads

Some DCE applications have direct dependencies on the following DCE services:

- Auditing
- Backing store
- Delegation
- ERA
- Event management
- Serviceability

Few DCE applications have direct dependencies on the following DCE services:

- Configuration
- Directory
- Host management
- Integrated login
- Password strength
- Registry

- Time
- UUID

No DCE applications have direct dependencies on the following DCE services because these services can be accessed only by means of the GSS-API or RPC security services:

- Authentication
- Protection

1.5.2 Indirect dependencies

An application indirectly depends on a DCE service if it uses the service by means of other DCE services (referred to here as *intermediary services*). This dependency can be removed by revising the application either to not use the intermediary services or to use replacement intermediary services that do not have these DCE dependencies. For example, an application indirectly depends on the DCE directory service if it makes calls to the DCE RPC namespace service. If the application is revised, for example, to use the JNDI protocol offered by IBM WebSphere Application Server as a replacement for the DCE RPC namespace service, the application no longer has this indirect dependency on the DCE directory service because IBM WebSphere Application Server does not have any DCE dependencies.

Many DCE applications have indirect dependencies on the DCE services, as shown in Table 1-2.

Table 1-2 Indirect dependencies for many DCE applications on DCE services

Service	Reason for indirect dependency
Authentication	All DCE servers use the RPC security service for authentication.
Authorization	All DCE servers use the authorization service.
Configuration	Some DCE services require a valid DCE configuration.
Directory	The RPC namespace service uses the RPC namespace database, which is maintained by the directory service.
Host management	The RPC endpoint service uses the endpoint database, which is maintained by the host management service.
Login	All DCE servers use the login service. Also, the DCE authentication and PAC services require the initial authentication information produced by the login service.

Service	Reason for indirect dependency
Messaging	All DCE services use the messaging service to report errors. Also, international DCE applications require the message service to separate messages for translation.
PAC	All DCE servers use the authorization service and implement this service in a way that requires the PAC.
Protection	All DCE servers use the RPC security service for protecting the integrity of data. The RPC security service performs this service by using the protection service.
Registry	The DCE login, authentication, and PAC services require the registry database, which is maintained by the registry service.
RPC basics	All DCE servers use the RPC basics service.
RPC endpoint	All DCE servers use the RPC endpoint service.
RPC namespace	All DCE servers use the RPC namespace service. Also, the DCE registry service uses the RPC namespace service.
RPC security	All DCE servers use the RPC security service.
Serviceability	All DCE servers use the serviceability service.
Threads	All DCE servers use the threads service.
Time	The time in the DCE cell must be synchronized if any of the DCE applications in the cell depend on the DCE services that are time-sensitive.
UUID	The RPC basics and authorization services use the UUID service.

Some DCE applications have indirect dependencies on the DCE services shown in Table 1-3.

Table 1-3 Indirect dependency for some DCE applications on DCE services

Service	Reason for indirect dependency
Auditing	The DCE security, directory, and time servers optionally can be configured to use the auditing service.
Backing store	The authorization service optionally can use the backing store service to store the ACL database.
ERA	The login and authentication services use ERAs to store and retrieve optional parameters. The password strength server uses ERAs to store password rules.

Service	Reason for indirect dependency
Host management	Some DCE applications require an environment in which administrators can configure remote hosts.
Password strength	Some DCE applications require an environment in which password rules are enforced.

Few DCE applications have indirect dependencies on the remaining DCE services, which are:

- Delegation
- Event management
- GSS-API
- Integrated login

1.5.3 No dependencies

An application does not have a dependency on a DCE service if the application does not directly or indirectly use the service. In this situation, there is no dependency to remove.

1.6 Summary of DCE review

DCE is middleware software used for creating, running, and managing distributed client and server applications (called *DCE applications*) in a heterogeneous environment. The users of DCE are developers, administrators, and sometimes end users. DCE consists of a set of services organized into components. Figure 1-4 lists each service and provides the following information about the service:

- component of the service
- purpose of the service
- users of the service (asterisk (*) indicates the primary user)
- frequency of DCE applications with direct dependencies on the service
- frequency of DCE applications with indirect dependencies on the service

Table 1-4 The dependencies of services

Service	Component	Purpose	Users	Direct	Indirect
Auditing	Security core	Audits events that might compromise security	Administrators* Developers	Some	Some

Service	Component	Purpose	Users	Direct	Indirect
Authentication	Security core	Invokes client and server authentication protocol	No direct users	None	Many
Authorization	Security core	Controls access to services based on configured permissions	Administrators* Developers	Many	Many
Backing store	Cross	Provides the use of a backend database	Developers	Some	Some
Configuration	Cross	Configures DCE servers	Administrators* Developers	Few	Many
Delegation	Security core	Lets servers assume the identities of clients	Developers	Some	Few
Directory	Directory	Provides the use of a common repository for information	Administrators* Developers	Few	Many
ERA	Security core	Extends the security registry database with user information	Administrators* Developers	Some	Some
Event management	Cross	Manages events	Developers	Some	Few
GSS-API	GSS-API	Uses GSS-API to involve the authentication service, protection service, or both	Developers	Many	Few
Host management	Cross	Manages remote hosts	Administrators* Developers	Few	Some
Integrated login	Security core	Configures a mapping between DCE and UNIX identities	Administrators* Developers	Few	Few
Login	Security core	Provides initial authentication information	End users* Developers Administrators	Many	Many

Service	Component	Purpose	Users	Direct	Indirect
Messaging	Cross	Displays the text associated with errors and system status in English and other written languages	Developers	Many	Some
PAC	Security core	Maps client identities to PACs	End users* Developers Administrators	Many	Many
Protection	Security core	Protects the privacy of data, the integrity of data, or both	No direct users	None	Many
Password strength	Security core	Configures password strength policy	Administrators* Developers	Few	Some
Registry	Security core	Provides the use of a security registry database	Administrators* Developers	Few	Many
RPC basics	RPC	Provides the basics for RPC	Developers*	Many	Many
RPC endpoint	RPC	Provides a way of storing and retrieving the ports of services	Developers	Many	Many
RPC namespace	RPC	Provides the use of a database for finding server host name and binding information	Developers* Administrators	Many	Many
RPC security	RPC	Uses RPC to invoke the authentication service, protection service, or both	Developers	Many	Many
Serviceability	Cross	Logs messages	Administrators* Developers	Some	Many
Threads	Threads	Provides compatibility library for developing multi-threaded applications	Developers	Many	Many
Time	Time	Synchronizes time	Administrators* Developers	Few	Many

Service	Component	Purpose	Users	Direct	Indirect
UUID	Cross	Generates universally unique identifiers	Developers	Few	Many

2

Replacement technologies

This chapter lists and recommends technologies that can be used to replace DCE services. The following topics are covered:

- Criteria used to select the technologies
- Technologies for C/C++ applications
- Technologies for Java applications

© Copyright IBM Corp. 2003. All rights reserved.

2.1 Criteria for selecting the technologies

The following criteria were used to select the recommended technologies:

- Compliance with industry standards
- Coverage of predominant DCE services
- Ease of migration
- Similarity to DCE services
- Technologies considered strategic
- Support of predominant platforms
- Support of predominant programming languages
- Availability of IBM implementations

2.1.1 Compliance with industry standards

Services offered by each recommended technology should conform to industry standards to help increase their usable life and enable interoperate with other software. Most of the recommended technologies meet this criterion.

2.1.2 Coverage of predominant DCE services

The set of recommended technologies should offer services that map to all of the DCE services on which many DCE applications have direct dependencies. See Section 1.5.1, “Direct dependencies” on page 12 for the names of these DCE services. The set of recommended technologies meets this criterion.

2.1.3 Ease of migration

It is desirable to have a method to migrate DCE data to a new technology. In some cases, this criterion is met, but in some other cases, additional design and administration tasks might be necessary. For example, IBM DCE Version 3.2 supports methods that can be used to migrate the contents of the DCE registry database to an open standard LDAP directory.

2.1.4 Similarity to DCE services

The services offered by each recommended technology should be functionally similar to the replaced DCE services. Many of the recommended technologies meet this criterion. For example, a replacement technology that supports the GSS-API requires little migration effort for those applications that use the GSS-API provided by DCE.

2.1.5 Technologies considered strategic

It is desirable that each technology is considered strategic by the industry and by IBM. This means that IBM is invested in researching and implementing the technology. Most of these technologies meet this criteria.

2.1.6 Support of predominant platforms

An implementation of each recommended technology should be available on the predominant platforms used in IBM DCE customer environments: AIX, Windows, and Solaris. Most of the recommended technologies meet this criterion.

2.1.7 Support of predominant programming languages

The IBM implementation of each recommended technology should have development tools that support the C language, which is the predominant programming language used by IBM DCE customers to develop and maintain DCE applications. Most of the recommended technologies meet this criterion. It also is desirable for development tools to support the Java language, as some IBM DCE customers might want to migrate their applications to Java. Some of the recommended technologies meet this criterion.

2.1.8 Availability of IBM implementations

It is desirable for each recommended technology to have an implementation of the technology that is provided and supported by IBM. Most of these technologies meet this criterion.

2.2 Technologies for C/C++ applications

The following technologies are recommended for applications that are written in the C/C++ language:

- aznAPI
- Common Object Request Broker Architecture (CORBA)
- DCE RPC basics
- DCE UUID
- Kerberos
- Lightweight Directory Access Protocol (LDAP)
- Network Time Protocol (NTP)
- Platform auditing
- Platform logging and messaging
- POSIX 1003.1c threads
- Web services

2.2.1 aznAPI

The *aznAPI* is a set of interfaces that developers can use to configure and control access to resources. The Open Group defined the aznAPI standard, and it can be seen at http://www.opengroup.org. aznAPI can be used to replace the DCE authorization, PAC, and UUID services. In many ways, aznAPI is similar to these services; the differences are:

- DCE: The application client sends its Kerberos ticket to a privilege server, the privilege server inserts a credential (a DCE PAC) into the ticket, the privilege server returns the ticket to the client, and the client sends the ticket with the PAC to the application server. The application server then receives the DCE PAC from the ticket and uses the DCE PAC information to determine authorization.
- aznAPI: On an "as needed" basis, the application server uses aznAPI to get a PAC for a given client and, using the PAC and a set of configured permissions, determines authorization. The application server cannot read the contents of the PAC.

IBM has implemented the aznAPI in the IBM Tivoli Access Manager product. IBM Tivoli Access Manager is available on many platforms including AIX, Linux, Solaris, and Windows. It provides interfaces in the C/C++ and Java programming languages. IBM offers an automatic method for migrating users, groups, and group membership data from IBM DCE Version 3.2 for AIX and Solaris to those platforms and versions of IBM Tivoli Access Manager that support storing this information in LDAP. This is possible because IBM DCE Version 3.2 for AIX and Solaris can be configured to store this registry data in LDAP using attributes shared with IBM Tivoli Access Manager. See "Option 1: IBM Tivoli Access Manager" on page 53 for more information.

A replacement strategy that uses IBM Tivoli Access Manager is described in "Option 1: IBM Tivoli Access Manager" on page 53. An example scenario and program that uses this replacement strategy can be found in Chapter 8, "Scenario 1: GSS-API application" on page 125.

Product information: The IBM Tivoli Access Manager line of products comprises three separate products:

- IBM Tivoli Access Manager for Business Integration
- IBM Tivoli Access Manager for e-business
- IBM Tivoli Access Manager for Operating Systems

The functionality described in this book that is used to replace the authorization function of DCE (that is, the implementation of the aznAPI with the underlaying support for IBM Directory Server and the administration interfaces) is part of the IBM Tivoli Access Manager for e-business.

The full IBM Tivoli Access Manager for e-business product supports additional functions, such as a Web authentication and authorization engine (called WebSEAL) and replication of its authorization databases. These additional functions are not of direct interest for the purpose of this book. Please refer to the product documentation or your local IBM representative for a full overview of the product's capabilities.

A comprehensive view of the IBM Tivoli Access Manager line of products, including other products, can be found online at:

http://www.ibm.com/software/tivoli/products

2.2.2 CORBA

CORBA is an object-oriented architecture and infrastructure defined by the Open Management Group (OMG). The purpose of CORBA is to enable computer applications to work together, regardless of the vendor, operating system, programming language, or network. In practice, however, implementations of CORBA are not always able to interoperate with each other, so it is recommended that you use different implementations of CORBA only if they have been tested against each other or against a common reference implementation.

CORBA consists of many components. The following list shows the five CORBA components that pertain to the subject of this book, plus the components' purpose and the OMG specifications that provide their standards. For the rest of this book, the term *CORBA* refers to the CORBA components listed here:

- *CORBA basics*: An architecture that permits a client object to invoke an operation on a local or remote server object. The architecture defines interfaces, protocols, and conventions including the CORBA interface definition language (IDL), the CORBA Object Request Broker (ORB), the CORBA General Inter-ORB Protocol (GIOP), and the CORBA Internet Inter-ORB Protocol (IIOP). For more information, refer to *The Common Object*

Request Broker: Architecture and Specification, available at:
http://www.omg.org

- *CosNaming*: An interface for the OMG Naming Service. The OMG Naming Service provides a mapping from names to object references. The document *Naming Service Specification* is available at the following Web site address:
http://www.omg.org/docs/formal/02-09-02.pdf
- *Common Secure Interoperability* Version 2 (CSIv2): A protocol for securing information transmitted over CORBA GIOP or IIOP. The protocol provides a way for the two parties to determine whether to use a more secure transport such as Secure Sockets Layer (SSL) or Transport Layer Security (TLS). It also defines message-level authentication protocols and a way for the two parties to determine whether to use a message-level authentication protocol. The document *The Common Security Interoperability Version 2 Specification* is available at http://www.omg.org/docs/ptc/01-06-17.pdf
- *C language mapping*: A mapping between the CORBA IDL and the C++ programming language. The document *C Language Mapping Specification* is available at the following Web site:
http://www.omg.org/technology/documents/formal/c_language_mapping.htm
- *Java language mapping*: A mapping between the CORBA IDL and the Java programming language. Two documents describe this mapping: The *Java Language Mapping to OMG IDL* and the *OMG IDL to Java Language Mapping*. They are available at the following two Web addresses (to be entered as single lines):
http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm
http://www.omg.org/technology/documents/formal/omg_idl_to_java_language_mapping.htm

An implementation of CORBA might be used to replace most of the services offered by DCE RPC. CORBA basics and mapping might replace the DCE RPC basics and endpoint services. CORBA CosNaming might replace the DCE RPC namespace service. CORBA CSIv2 might replace the DCE RPC security service.

Functionally, CORBA has many similarities to DCE RPC. The following is a list of differences:

- The IDL used by CORBA is object-oriented. The IDL used by the DCE RPC basics service is procedural-oriented.
- The CSIv2 used by CORBA makes it possible for two parties to negotiate the type of authentication protocol they will use or whether they will use the authentication protocol offered by the transport layer. DCE supports only the Kerberos authentication protocol.

IBM offers an implementation of CORBA in IBM WebSphere Application Server Version 5 or later (referred to hereafter as IBM WebSphere Application Server). The enterprise edition of this product provides Java and C++ development environments. The base edition provides a Java development environment only.

The CORBA C++ support provided in IBM WebSphere Application Server enables the use of CORBA interfaces between a server object providing a service and a client using the service. In practice, this means that two types of CORBA clients, WebSphere CORBA C++ clients and CORBA clients from vendors supported by WebSphere, can access two types of servers, CORBA C++ servers and WebSphere enterprise beans servers.

In addition, IBM WebSphere Application Server provides a basic CORBA environment that can “bootstrap” into the J2EE name space and invoke J2EE transactions. However, it does not provide its own naming and transaction services. Therefore, CORBA C++ clients and servers rely on WebSphere to provide these services.

WebSphere CORBA support can be divided into two categories:

WebSphere to WebSphere CORBA support

This enables the creation of CORBA client and server applications within the IBM WebSphere Application Server environment. You can use the CORBA C++ SDK to build a lightweight WebSphere CORBA C++ server to use with new or existing C and C++ applications. The lightweight CORBA C++ server supports the CORBA basics and namespace (CosNaming) protocols, but it does not support the CORBA security (CSIv2) protocol. You can also use the SDK to build a WebSphere CORBA C++ client to use with a WebSphere enterprise bean server or WebSphere CORBA C++ server. The CORBA C++ client and enterprise bean support the CORBA basics, CosNaming, and CSIv2 protocols. However, the CSIv2 support available with CORBA C++ clients is limited to using SSL transport for authentication.

Product information: The CORBA C++ SDK ships with IBM WebSphere Application Server Enterprise, Version 5.

WebSphere to other ORB support

This enables other applications that are based on CORBA Object Request Brokers (ORBs) to interoperate with WebSphere. It enables these applications to leverage WebSphere-supported open technologies, such as Java Server Pages, XML, Java servlets, and enterprise beans.

For customers that require a full C++ CORBA development environment, some independent vendor products are available that might meet these requirements.

The Java development environment is discussed in Section 2.3.4, “IBM implementation of J2EE: WebSphere Application Server” on page 39.

A sample application migration scenario that makes use of the CORBA replacement technology is further detailed in Chapter 9, “Scenario 2: Non-secure RPC application” on page 171.

2.2.3 DCE RPC

The source code for the DCE RPC technology can be downloaded from The Open Group and could be used to create a stand-alone offering of the DCE RPC technologies. This would require a considerable amount of development effort and currently no such offering is available as a product. However, if such an offering were made available, it might serve as a valuable option to use in replacing the DCE RPC services. Therefore, this book discusses stand-alone DCE RPC as a possible technology that could be used to replace the DCE RPC services.

2.2.4 DCE UUID

The DCE UUID technology is used by the DCE RPC and authorization services. This redbook recommends replacement technologies for both of these services. However, some organizations might have a large investment in user-written authorization code and want to continue using this code for authorization rather than replacing it with a new technology. These customers might find it helpful to obtain the DCE UUID code, with which they can generate new DCE UUIDs to use with their user-written authorization code. The source for the DCE UUID code can be downloaded from The Open Group (`http://www.opengroup.org`).

2.2.5 Kerberos

Kerberos Version 5 (called *Kerberos* hereafter) is a network authentication technology that was developed by the Massachusetts Institute of Technology (MIT) and implemented by many vendors including IBM, Microsoft, SuSE, Sun, Hewlett Packard, and CyberSafe. Implementations of Kerberos are available on many platforms including AIX, OS/390, OS/400®, Linux, Windows 2000, Solaris, and UNIX. The following industry standards have been set for Kerberos:

- The over-the-wire protocol that Kerberos uses for logon, authentication, and protection is defined in Internet Engineering Task Force (IETF) Request for Comments (RFC) 1510.
- The GSS-API interfaces and over-the-wire protocol are defined in IETF RFCs 2743 and 2744, and the Kerberos implementation of GSS-API is defined in IETF RFC 1964.

Kerberos can be used to replace the following DCE services:

- Authentication
- Delegation
- Login
- GSS-API
- Protection

Functionally, Kerberos performs these services in a way that is similar to DCE. This is because the DCE implementations of these services are based on Kerberos. The few functional differences have to do with the following features that DCE added to the Kerberos protocol:

- DCE third-party and public key pre-authentication protocols are not supported by the Kerberos standard, as specified in RFC 1510. IETF is working on a proposed standard for a Kerberos public key pre-authentication protocol, but this is not yet a standard at the time of writing.
- The DCE sealed delegation is not supported by the Kerberos standard, which only supports impersonation delegation.

Although Kerberos is functionally similar to DCE, most of its interfaces are different. The only Kerberos interfaces that are the same as those used by DCE are the GSS-API interfaces defined in IETF RFC 2743 and 2744. The remaining Kerberos interfaces are different syntactically from those interfaces used by DCE.

The IBM implementation of Kerberos is named *IBM Network Authentication Service*. IBM Network Authentication Service supports the following platforms in the C development environment:

- AIX Version 5.1 and Version 5.2
- OS/390 Release 10
- OS/400 Version 5 Release 1 (client only)

IBM offers an automatic method for migrating authentication data (principals and policies) from IBM DCE Version 3.2 for AIX and Solaris to those platforms and versions of IBM Network Authentication Service that support storing this data in LDAP. This is possible because IBM DCE Version 3.2 for AIX and Solaris can be configured to store this data in LDAP using attributes shared with IBM Network Authentication Service. See section 3.1.6, “Delegation, GSS-API, and login” on page 56 for more information.

Chapter 8, “Scenario 1: GSS-API application” on page 125 explains the use of IBM Network Authentication Service as a replacement technology in a practical example.

Product information: The IBM Network Authentication Service is not a separate IBM product. It is available with current levels of IBM AIX, as well as other IBM operating systems.

2.2.6 LDAP

LDAP Version 3 (called *LDAP* hereafter) is a technology for accessing and updating directory information. LDAP is a lightweight derivative of the X.500-based directory services and its Directory Access Protocol (DAP). Although, strictly speaking, the term *LDAP* refers to a protocol geared for accessing directory information, the term is frequently used in similar, sometimes confusing contexts. It is common to refer to a directory that supports LDAP as an *LDAP directory* or to a system that runs such a directory as an *LDAP system*.

The information that is stored in an LDAP directory is organized in a hierarchical, inverted-tree form. At the top of such an *Directory Information Tree* (DIT) are one or more suffixes, which are one of the first steps an administrator must define when configuring an LDAP directory. Following are two typical forms of such suffixes as they may appear in LDAP directories (notice that a suffix may already contain several elements that may be confused with a certain level of the DIT):

- *dc=IBM,dc=com* (*dc* stands for domain component)
- *ou=Sales,o=IBM,c=US* (*ou*, *o*, and *c* stand for organizational unit, organization, and country, respectively)

Information is stored in an LDAP directory in objects and attributes. For example, information about an employee may be stored as a single object that represents the employee's name and that has several attributes that store information such as employee number, date of birth, telephone number, and the like. To continue with the example, Joe Smith may be represented in an LDAP directory as *cn=Joe Smith,ou=Sales,o=IBM,c=US* (*cn* stands for common name). The full reference to Joe Smith as shown is called its *Distinguished Name* (DN), which must be unique in the directory. There might be several Joe Smiths in the same directory, all starting with *cn=Joe Smith*, but their full DN must be distinct. Thus, *cn=Joe Smith* is also called the object's *Relative Distinguished Name* (RDN).

The LDAP protocol also defines powerful search and retrieve operations on directory data. However, it does not define how the objects are arranged in the DIT and what information the directory may contain. In fact, an LDAP directory may be used to store a variety of information, including binary files, and it is a common misbelief that an LDAP directory is primarily used for storing people-related information. In the course of this book, LDAP directories will be used for storing security-related information as well as authorization data.

The type of information that can be stored, as well as any rules that go along with the information types, are defined in LDAP schema. An LDAP directory product, such as the IBM Directory Server, comes with a default schema that fits most uses. If the default schema is not sufficient (for example, if specific information such as DCE registry information is to be stored), the schema can be modified, as we will see later in this book.

The details of the LDAP protocol are defined in IETF RFC 2251. The following industry standards have been defined for LDAP:

- IETF RFC 1274 The COSINE and Internet X.500 Schema
- IETF RFC 1777 Lightweight Directory Access Protocol (Version 2)
- IETF RFC 1778 String Representation of Standard Attribute Syntaxes
- IETF RFC 1779 String Representation of Distinguished Names
- IETF RFC 1823 LDAP Application Program Interface (Version 2)
- IETF RFC 2052 A DNS RR for Specifying the Location of Services (DNS SRV)
- IETF RFC 2219 Use of DNS Aliases for Network Services
- IETF RFC 2222 Simple Authentication and Security Layer (SASL)
- IETF RFC 2247 Using Domains in LDAP/X.500 Distinguished Names
- IETF RFC 2251 Lightweight Directory Access Protocol (Version 3)
- IETF RFC 2252 Lightweight Directory Access Protocol (Version 3): Attribute Syntax Definitions
- IETF RFC 2253 Lightweight Directory Access Protocol (Version 3): UTF-8 String Representation of Distinguished Names
- IETF RFC 2254 The String Representation of LDAP Search Filters
- IETF RFC 2255 The LDAP URL Format
- IETF RFC 2256 A Summary of the X.500(96) User Schema for use with LDAP Version 3
- IETF RFC 2891 LDAP Control Extension for Server Side Sorting of Search Results

LDAP might be used to replace the following DCE services:

- Registry
- Directory
- Backing store

Functionally, LDAP offers some similarities to the DCE directory service. The IBM implementation of LDAP is the *IBM Directory Server*. IBM Directory Server (formerly named IBM SecureWay® Directory) is available on these platforms:

- AIX
- HP-UX
- Linux
- Solaris
- Windows

The development support for IBM Directory Server is provided in the C language. Also, the IBM Directory Server can be accessed through the Java language using the Java Naming and Directory Interface (JNDI). See 2.3, “Technologies for Java applications” on page 34.

IBM DCE Version 3.2 for AIX and Solaris provides a way to migrate the DCE registry database to LDAP. IBM does not provide a way to migrate backing store, directory, or RPC namespace data to LDAP. For more information, see the *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.

Product information: The IBM Directory Server is available at no charge for the platforms as listed above, as well as for the OS/400 and OS/390 operating system platforms.

For the AIX, HP-UX, Linux, Solaris and Windows platforms, download the IBM Directory Server from the Downloads link at:

`http://www.ibm.com/software/network/directory/server`

For documentation about the IBM Directory Server, go to the same Web page and click the **Library** link.

2.2.7 Network Time Protocol

NTP Version 3 is a technology used for synchronizing time among a set of distributed time servers and clients. It is available on many platforms, including Solaris, HP-UX, Ultrix, OSF/1, IRIX, AIX, A/UX, PTX®, FreeBSD, NetBSD, BSD/386, Linux, and Unixware. The standard for NTP is defined in IETF RFC 1305. This standard makes it possible for different implementations of NTP to interoperate over-the-wire and to use the same architecture and algorithms for synchronizing time. NTP servers can interact with SNTP (Simple Network Time Protocol), which is available on many Windows platforms.

NTP might be used to replace the DCE time service. The configuration of NTP and DCE time service has many similarities.

IBM offers an implementation of NTP called the *AIX xntpd daemon* on AIX platform versions 4.2 and later. The xntpd daemon can be accessed only through command interfaces. DCE data does not need to be migrated to NTP.

2.2.8 Platform auditing

All operating system platforms that have been C2 certified by the National Security Agency (NSA) offer a set of auditing interfaces that can be accessed programmatically from a C/C++ program or administratively through either the command line or an administration tool. The AIX, Solaris, and Windows platforms all are C2 certified, so each of these platforms offers auditing interfaces. These interfaces might be used to replace direct dependencies to the DCE auditing service.

There are no standards for auditing interfaces, so they differ from one platform to another and from the DCE auditing interfaces. However, if the platform is C2 certified, its auditing interfaces must allow application developers and administrators to do all of the auditing functions required for C2 certification. Therefore, the functions provided by the auditing interfaces of all C2-certified platforms are similar. These functions also are similar to the functions provided by the DCE auditing interfaces, as the DCE auditing interfaces were designed to meet C2 certification requirements.

2.2.9 Platform logging and messaging

Most operating system platforms offer a set of logging and messaging interfaces that can be accessed programmatically from a C/C++ program or administratively from either the command line or an administrative tool. The following logging and messaging interfaces are available on UNIX and Windows platforms.

UNIX logging and messaging

The UNIX logging (syslog()) and messaging standards are defined in The Open Group UNIX 98, POSIX.1-1996 standards. These standards can be used as partial replacements for DCE messaging and serviceability.

UNIX offers the syslog() function, which provides a logging service similar to the logging service provided by DCE serviceability. The logging scheme provided by the syslog() function does not automatically fetch a status code’s error text from a message catalog as DCE serviceability does, but it does support a similar type of facility-based logging. For an example of how to use the syslog() function on UNIX platforms, see the *AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions, Volume 2* at the following Web site:

http://publibn.boulder.ibm.com/cgi-bin/ds_form?lang=en_US&viewset=AIX

UNIX messaging is similar to DCE messaging in that both use message catalogs and the XPG4 locale standard defined by The Open Group. The difference between the two messaging services is that DCE messaging supports unique, system-wide error codes, but UNIX messaging does not provide this support. See chapter 16 about national language support in the *AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs*, available at:

http://publibn.boulder.ibm.com/cgi-bin/ds_form?lang=en_US&viewset=AIX

IBM implementations of UNIX exist on all IBM hardware. Because UNIX uses XPG4 message catalogs, message catalog files can be migrated from DCE to UNIX.

Windows logging and messaging

Windows platforms provide logging and messaging services that can be used as partial replacements for the DCE serviceability and messaging services. For logging, Windows supports proprietary logging to the Windows Event Log using functions such as RegisterEventSource() and ReportEvent(). When used in conjunction with message resource files, Windows can log message text that is translated into the international language that matches the Windows regional settings. Search for *Event Logging Operations* to get an overview of Windows event logging and the associated API at:

http://msdn.microsoft.com/

Windows messaging is provided through proprietary support for message resource files and functions that enable the support of multiple international languages based on the Windows regional settings. Windows messaging is implemented somewhat differently from the way DCE messaging is implemented. See *Writing Multilingual User Interface Applications* at the following Web site for a discussion and examples of how to use the Windows proprietary messaging support:

http://www.microsoft.com/globaldev/articles/articles.asp

2.2.10 POSIX 1003.1c threads

POSIX 1003.1c, draft 10 is an Institute of Electrical and Electronics Engineers (IEEE) standard defining a set of APIs, often called *pthreads*, for writing multi-threaded applications. This set of APIs has evolved into a standard for operating systems. Therefore, implementations of POSIX 1003.1c draft 10 are part of many operating system platforms, including AIX, Solaris, HP-UX, and Linux. Windows threads do not conform to POSIX 1003.1c, draft 10, but some third parties provide libraries that map POSIX 1003.1c threads to Windows threads.

POSIX 1003.1c, draft 10 might be used as a replacement for the DCE threads service. The DCE thread service is based on POSIX 1003.1c, draft 4 (also called *POSIX 1003.4a*) and is sometimes called *Concert Multi-thread Architecture (CMA) threads*. The semantics of most pthreads calls defined in POSIX 1003.1c, draft 10 are the same as those used by the DCE thread service. Differences are:

- Syntax changes to some APIs
- Return error codes are used instead of setting the global *errno* variable for all pthread functions
- New functions are introduced to enhance thread cancellation and thread scheduling
- Differences in signal handling
- Specification of cancellation points
- Changes to the reentrant C library API

DCE data does not need to be migrated to POSIX 1003.1c.

2.2.11 Web services

Web services are self-contained application modules that you can describe, publish, locate, and invoke over a network. A Web service is described using a standard, formal eXtensible Markup Language (XML) notion, called its service description. It covers all of the details necessary to interact with the service, including message formats (that detail the operations), transport protocols, and location. At minimum, Web services consist of a service requester, a service provider, and a service registry. The service provider publishes its Web service in a service registry. The service requester finds the Web service in the service registry and invokes the service by binding to the service provider.

The standards for these operations are:

- World Wide Web Consortium (W3C) *Simple Object Access Protocol* (SOAP) Version 1.1: Defines the XML messages that are exchanged between the service requester and service provider when the service is invoked. These messages can be transmitted over any protocol, such as HTTP and TCP/IP.
- W3C *Web services Description Language* (WSDL) Version 1.1: Defines information about the service. The service provider publishes this information in the service registry and the service requester finds this information in the registry and uses it to invoke the service.
- uddi.org *Universal Description Discovery and Integration* (UDDI) Version 3.0: Defines the protocol used to publish and find information in the service registry.

Web services might be used to replace most of the services offered by the DCE RPC component. SOAP and WSDL might be used as replacements for the DCE RPC basics and DCE RPC endpoint services. They handle serialization, marshalling, and bind protocols. The UDDI might be used to replace the RPC namespace service.

A specification for SOAP security is being defined by various vendors through the OASIS organization. The specification proposes a set of SOAP extensions for implementing security by means of a variety of security models including PKI, Kerberos, and SSL.

IBM provides C/C++ development support for SOAP in the form of toolkits. At the time of writing, these toolkits are in the form of technical previews. IBM also provides Java development support for SOAP on some platforms of WebSphere Application Server.

For more information, refer to the following Web sites:

- For more information about the XML Version 1.0 standard, see `http://www.w3.org/TR/REC-xml`
- For more information about the SOAP Version 1.1 standard, see `http://www.w3.org/TR/SOAP/`
- For more information about the WSDL Version 1.1 standard, see `http://www.w3.org/TR/wsdl`
- For more information about the UDDI Version 3.0 standard, see `http://www.uddi.org/specification.html`
- For more information about the SOAP security specification, see `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`

2.3 Technologies for Java applications

The recommended technology for applications being migrated to the Java language is the Java 2 Enterprise Edition Version 1.3 or later (called *J2EE* hereafter). J2EE is a Java platform specification for developing, running, and managing applications in an enterprise environment. Implementations of J2EE are available on most operating system platforms.

2.3.1 J2EE application environment

J2EE defines a platform for running the following types of components:

- Java applets
- Java client applications

- Servlets
- Java Server Pages (JSPs)
- Enterprise beans

Java applets and *Java client applications* are both client applications that are written in the Java programming language. Java applets are downloaded dynamically from a Web server to a client machine and run on the client machine. Java client applications are installed and run on the client machine.

Servlets, *JSPs*, and *enterprise beans* are all server applications. Servlets are written in the Java programming language and extend the function of a Web server, typically by generating dynamic content and interacting with clients using HTTP. JSP files are the same as servlets except that they are written in the Java scripting language and then converted to a servlet when they are run. A JSP file is referred to hereafter as a servlet. Enterprise beans are written in the Java programming language and implement a business task.

Each type of component is run in a *container*. The container is configurable and handles system tasks, such as network communications and security. For example, an enterprise bean is run in an enterprise bean container, which is configurable and handles network communications, security, multi-threading, and other system functions. The container simplifies programming for the developer and makes it possible for an administrator to revise the configuration of system tasks at run time.

Some component types can interoperate with Web browsers, legacy applications, or both. Servlets can interoperate with Web browsers using the HTTP protocol. Enterprise beans can interoperate with legacy client programs using CORBA protocols. Java client applications, servlets, and enterprise beans can interoperate with legacy server programs using either CORBA protocols or the Java Connector Architecture (JCA).

2.3.2 Standards for the J2EE

Implementations of J2EE must provide a platform for running each component type. The platform must conform to the following standards.

J2EE deployment specification

A standard that defines a common way of packaging applications. This standard, referred to as a document type definition (description language (DTD)), is defined in the Java 2 Platform Enterprise Edition Specification Version 1.3.

Java technology standards for the J2EE platform

The Java technology standards are a set of specifications that are defined by the Java organization. These specifications include the following:

- Java 2 Platform Enterprise Edition Specification Version 1.3
- Java 2 Platform Standard Edition (J2SE) Version 1.3 API Specification
- Java Database Connectivity (JDBC) Version 2.0 Standard Extension API
- Java Naming and Directory Interface Version 1.2 Specification
- JDBC Version 2.1 Application Programming Interface (API)
- Enterprise JavaBeans (EJB) Specification, Version 2.0
- Java Servlet Specification, Version 2.3
- Java Server Pages Specification, Version 1.2
- Java Message Service (JMS), Version 1.0
- Java Transaction API (JTA), Version 1.0.1
- Java Transaction Service, Version 1.0
- JavaMail API Specification Version 1.2
- JavaBeans Activation Framework (JAF) Specification Version 1.0
- Java API for XML Parsing (JAXP) Version 1.1
- Java Connector Architecture (JAC) Version 1.0
- Java Authentication and Authorization Service (JAAS) Version 1.0

IETF and de facto standards for the J2EE platform

The set of standards that is defined by the IETF and a de facto standard that is defined by Netscape include:

- IETF RFCs defining the TCP/IP protocol family
- IETF RFC defining HTTP Version 1.0
- IETF RFC 2246: The TLS Version 1.0 Protocol
- IETF RFC defining HTML Version 3.2
- Netscape SSL Version 3.0

CORBA standards for the J2EE platform

CORBA standards include a set of standards defined by Object Management Group:

- CORBA Version 2.3.1 Specification
- IDL to Java Language Mapping Specification
- Java Language to IDL Mapping Specification
- CORBA CosNaming Service
- Interoperable Naming Service
- Common Secure Interoperability Version 2 Final Available Specification

2.3.3 DCE services that can be replaced by J2EE

The J2EE standards listed in the previous section specify the APIs and functions that must be implemented by J2EE components. Some of these APIs and functions might be used to replace the following DCE services:

- Authentication
- Authorization
- Backing store
- Delegation
- Directory
- Login
- Messaging
- PAC
- Protection
- RPC (all RPC services)
- Threads

Therefore, a J2EE implementation might be used to replace most, but not all, of the DCE services. (For a complete list of DCE services, see section 1.6, "Summary of DCE review" on page 15). The following describes the J2EE specification for each set of J2EE APIs or functions that can be used to replace a DCE service.

Authentication and login

All containers for Java applets, Java client applications, and servlets must support the following methods of transmitting client identity information:

- HTTP basic authentication (client passes its user identity and password)
- HTTP form-based authentication (client passes its user identity and password in an HTML form)
- HTTPS (client passes its public certificate using an SSL transport)

If SSL is used (as it is with HTTPS), the servlet transmits its identity information (a public certificate) to the client.

In addition, all Java client application, servlet, and enterprise bean containers must support CSIv2 as described in "RPC" on page 38.

Authorization

All servlets and enterprise beans must support the configuration of roles and the mapping of these roles to operating data, such as users and groups. In addition, all servlet and enterprise bean containers must support declarative and programmatic authorization. With declarative authorization, an administrator configures authorization to servlets and methods in enterprise beans based on

roles. With programmatic authorization, the developer uses the Java authorization API, which is a set of APIs consisting of isUserInRole and getUserPrincipal for servlets and isCallerInRole and getCallerPrincipal to program authorization for enterprise beans.

Backing store

All containers for servlets and enterprise beans must support the JDBC API, which is a set of Java APIs for accessing structured query language (SQL) databases and other databases that use tabular data.

Delegation

All containers for servlets and enterprise beans must support the propagating of a client credential so that an impersonation type of delegation can be performed.

Directory

All Java applets, Java client applications, servlets, and enterprise beans must support the JNDI, which is a set of APIs for interfacing with a directory.

Messaging

All Java applets, Java client applications, servlets, and enterprise beans must support the Java String, ResourceBundle, and Locale APIs. The String API can be used to print messages. The ResourceBundle and Locale APIs can be used to support multiple languages. See the *Java Internationalization Tutorial* at `http://java.sun.com/docs/books/tutorial/i18n/index.html` for a discussion and examples of how to handle messages in a Java application.

PAC

All containers for servlet and enterprise beans must be able to map a client identity to a credential and associate the credential with the client context. The credential contains a set of attributes. The format of the credential and the set of attributes contained in the credential is implementation-specific.

Protection

All containers for Java applets, Java client applications, servlets, and enterprise beans must support the ability to use SSL for transport. SSL protects transmitted data with encryption and checksum algorithms.

RPC

All containers for Java client applications, servlets, and enterprise beans must support the client side of Remote Method Invocation over Internet Inter-Orb Protocol (RMI over IIOP), and all enterprise bean containers must support the server side of RMI over IIOP. RMI over IIOP is a combination of Java and CORBA

protocols that enables clients to invoke methods on remote objects. This is done as follows: the client finds the method using either Java JNDI or CORBA CosNaming; invokes the method using either Java RMI or CORBA IDL; serializes, marshals, and transmits the method data using Common Object Request Broker Architecture over Internet Inter-Orb Protocol (CORBA IIOP); and, if required, secures the transmission using CORBA CSIv2. The method data is transmitted in Unicode, so multiple languages are supported.

CSIv2 must be implemented as specified in Conformance Level 0. In this conformance level, CSIv2 must be able to pass client identity information using basic authentication (client passes its user identity and password), client certificate authentication (client passes its public certificate using an SSL transport), or credential (client passes its credential).

Threads

All Java applets, Java client applications, and servlets must support the Java Threads API, which is a set of APIs for developing multi-threaded applications. All enterprise bean containers must handle multi-threaded, thread safe logic. Because enterprise bean containers handle the threading logic, J2EE requires enterprise bean containers to prevent enterprise beans from using the Java Threads APIs.

2.3.4 IBM implementation of J2EE: WebSphere Application Server

The IBM implementation of J2EE is IBM WebSphere Application Server Version 5 (called *WebSphere Application Server* hereafter). WebSphere Application Server is available on many platforms including AIX, Linux, OS/400, OS/390®, Windows, and Solaris.

IBM WebSphere Application Server can be used to replace all of the DCE services described in the previous section. In addition, IBM WebSphere Application Server offers extensions that map to DCE services:

- A registry database, which can be configured to be an LDAP directory, a local operating system, or a custom database.
- An authentication mechanism, which validates the authentication information transmitted by the client against the data configured in the registry database and generates a client credential containing the attributes of the client. For example, with the LTPA mechanism, if the client transmits a user identity and password, LTPA validates this identity and password against the user identity and password configured in the registry database. It then generates a credential similar to a DCE PAC containing the user identity of the client and a listing of the groups in which the client is a member.
- A method for configuring a login context for clients accessing servlets.

IBM offers an automatic method for migrating users, groups, and group membership data from IBM DCE Version 3.2 for AIX and Solaris to those platforms and versions of IBM WebSphere Application Server that support storing this information in an LDAP directory. This is possible because IBM DCE Version 3.2 for AIX and Solaris can be configured to store this data in an LDAP directory using attributes shared with IBM WebSphere Application Server. See section 3.2.2, “Revising the application environment for the new architecture” on page 72 for more information.

The WebSphere Application Server extensions as described above are used and demonstrated in two sample scenarios in this book.

2.3.5 Additional information on IBM WebSphere Application Server

For your convenience, this section lists and explains more features of the WebSphere Application Server for better understanding of the similarities to and differences from a DCE environment. The information provided in this section repeats and expands on the general J2EE discussion of the previous sections.

WebSphere login and authentication mechanisms

When an application client requires access to a protected enterprise bean, it sends the authentication information to the WebSphere Application Server using the CSIv2 or SAS protocol. These are add-on IIOP services, where IIOP is the application communication protocol. The underlying transport layer protocol can be TCP/IP with or without SSL. Figure 2-1 depicts the relationship between these protocols.

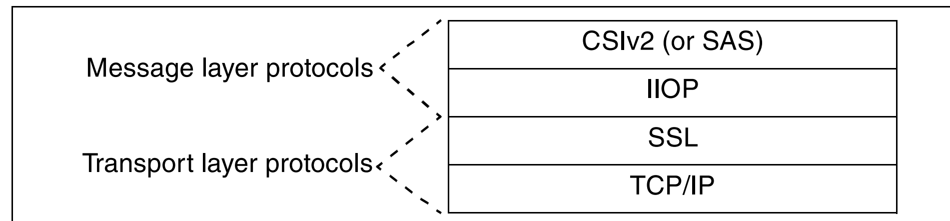


Figure 2-1 Authentication protocols

The authentication information sent by the client can be either basic authentication (user ID and password), credential token, or client certificate. The EJB Security Collaborator passes this data on to the Authentication Module that is implemented using the Java Authentication and Authorization Service (JAAS) login module. The login module can be either Lightweight Third Party Authentication (LTPA) or (Simple WebSphere Authentication Mechanism (SWAM).

The authentication module uses the registry that is configured on the system to perform authentication. The registry can be either the local operating system (LocalOS), LDAP, or a custom registry (CustomRegistry). Once the user is authenticated, the login module returns the user credentials to the EJB Security Collaborator in the EJB Container, which is used by the Authorization Engine to perform further access control checks.

WebSphere authorization mechanisms

The WebSphere Application Server standard authorization mechanisms are based on the J2EE security specification and JAAS. JAAS extends the security architecture of the Java 2 Platform with additional support to authenticate and enforce access controls upon users.

The JAAS programming model enables the developer to design application authentication and authorization in a pluggable fashion, which makes the application independent from the underlying authentication and authorization technology. The Java 2 security architecture uses security policy to specify who is allowed to execute a piece of code of the application. Code characteristics, such as a code signature, signer ID, or source server, decide whether the code will be granted access to be executed or not. JAAS extends this approach with role-based access control.

The J2EE specification defines a security role as "a logical grouping of users that are defined by an Application Component Provider or Assembler." Security roles provide a mechanism whereby application developers determine the security policies for an application by creating named sets of users (such as managers, customers, and employees) that will have access to secure resources and methods. At application assembly time, these sets of users, or security roles, are not tied to any real users or groups of users. Instead, they are placeholders that will be mapped to real users and groups at application deployment time, during a process called *security role mapping*. The J2EE Containers are responsible for enforcing access control on component objects and methods. Containers provide two types of security: *declarative security* and *programmatic security*.

Declarative security

Declarative security is the means by which an application's security policies can be expressed externally to the application code. At application assembly time, security policies are defined in an application's deployment descriptor. A deployment descriptor is an XML file that includes a representation of an application's security requirements, including the application's security roles, access control, and authentication requirements. When using declarative security, application developers are free to write component methods that are completely unaware of security. By making changes to the deployment

descriptor, an application's security environment can be changed radically without requiring any changes in application code.

Programmatic security

Programmatic security is used when an application must be "security aware." For instance, a method might need to know the identity of the caller for logging purposes, or it might perform additional actions based on the caller's role. The J2EE specification provides an API that includes methods for determining both the caller's identity and role. The enterprise bean methods are: *isCallerInRole* and *getCallerPrincipal*.

Please note that WebSphere also can be configured to use external authorization systems, such as IBM Tivoli Access Manager. Please refer to the IBM Redbook *IBM WebSphere V5.0 Security* for more details.

WebSphere delegation

Delegation enables an intermediary enterprise bean to perform a task initiated by a caller under the identity of that very caller or another specific security identity. The WebSphere Application Server product provides delegation by implementing the *run-as* feature of the J2EE specification. Run-as rules can be applied on enterprise bean level or on enterprise bean method level. LTPA tokens are used to carry run-as identities to downstream servers.

The *run-as* feature is comparable to the *impersonation delegation* feature of DCE. With DCE impersonation, an intermediate server is allowed to impersonate only the original caller, if that one allows for it.

WebSphere Application Servers also support CSIv2 *identity assertion*. However CSIv2 *identity assertion* is not really identity delegation, as it does not provide the downstream server(s) with means to authenticate the asserted identities.

For the sake of completeness, it is worth mentioning that DCE also provides *sealed delegation*. With DCE sealed delegation, each downstream server receives the credentials of all intermediate servers in the upstream chain. WebSphere Application Server does not provide this type of delegation.

WebSphere protocols: RMI over IIOP

Figure 2-2 on page 43 gives an overview of how a CORBA or J2EE application client binds to an Application Server in the J2EE model. All communications from the application client to the Application Server use RMI over IIOP. RMI over IIOP is the RPC mechanism that can replace DCE RPC. RMI over IIOP, unlike pure RMI, enables J2EE applications to communicate with CORBA clients and servers.

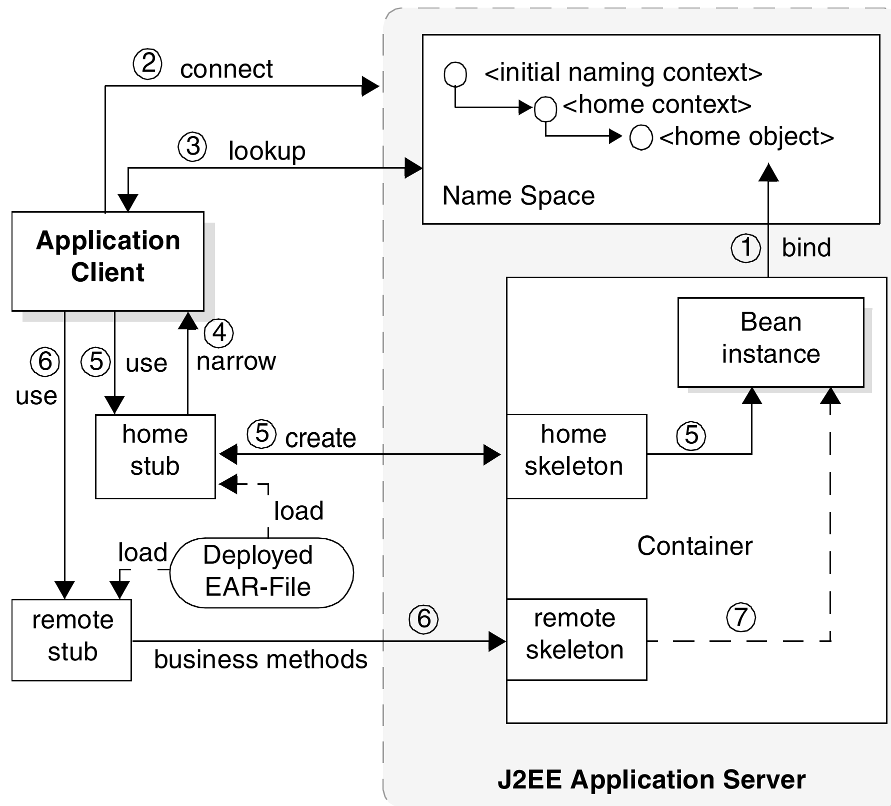


Figure 2-2 J2EE connection process using RMI over IIOP

The different communications that take place in a J2EE connection process are described in reference to the numbers in Figure 2-2:

1. At startup time, among other activities, the J2EE application binds the IOR to a home object with the Application Server's name service, using the JNDI interface.
2. When starting up, the application client makes an initial connection to the Application Server.
3. The application client uses JNDI or CosNaming to look up the IOR to the desired home interface of the enterprise bean.
4. The application client uses the IOR, obtained in the previous step, and narrows it down to the home object of the enterprise bean. The home stub is the client side implementation of the bean's home interface.
5. In the next step, the application client calls the home stub's `create` method to obtain an object reference to the corresponding remote interface of the

enterprise bean. On the server side, an instance of the desired enterprise bean is created and the `create` method of that enterprise bean is invoked. After successful completion, an object reference to the newly created bean instance is returned to the application client. The remote stub is the client side implementation of the bean's remote interface.

6. In this step, the application client uses the object reference to the enterprise bean's remote interface to call business methods exposed in this interface.
7. A request to a method of the enterprise bean's remote interface is not routed directly to the corresponding enterprise bean instance, but is received by the J2EE Application Servers container, in which the enterprise bean is deployed. After performing container-specific duties, such as checking client access or performing enterprise bean activation, the request is finally forwarded to the enterprise bean instance.

WebSphere network deployment

Figure 2-3 on page 45 shows an overview of IBM WebSphere Application Server network deployment configuration. Although this is not of particular importance for understanding DCE replacement strategies, it is interesting to note that the WebSphere network deployment uses the term *cell*, which is not to be confused with a DCE cell. A cell in the WebSphere network deployment is a grouping of nodes into a single administrative domain. A node is a logical grouping of WebSphere managed server processes that share common configuration and operational control. The Application Servers inside each node provide the run-time environment for the enterprise application.

For each Java application component, the WebSphere Application Server provides containers (namely a Web container, EJB container, and JCA container) to enable the execution of the components. The Web container processes servlets, Java Server Page (JSP) files, and other types of server-side includes. The EJB container provides an interface between enterprise beans and the Application Server. It provides all of the run-time services to deploy and manage enterprise beans.

The WebSphere Deployment Manager provides a single point of administrative control for all elements in the cell. The node agent on each node coordinates with the deployment manager to synchronize the configuration and to perform management operations on behalf of the deployment manager.

The application client container is a separately installable component on the client machine(s), and it hosts J2EE clients. These clients communicate with the enterprise applications deployed on the WebSphere Application Servers.

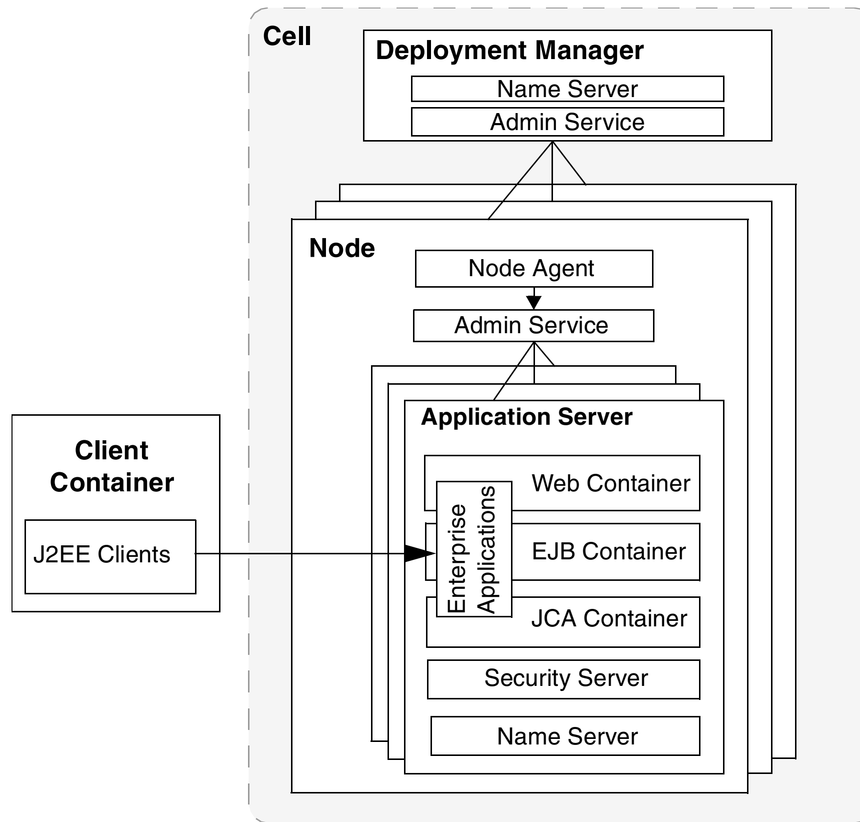


Figure 2-3 IBM WebSphere Application Server architectural overview

WebSphere naming service

The WebSphere naming implementation is composed of Java Naming, Directory Interface (JNDI) and CosNaming. JNDI provides the client-side access to naming, presents the programming model used by application developers, and is essentially built on top of CosNaming. CosNaming provides the server-side interface and is where the name space is actually stored. The name space can be accessed and manipulated through a name server.

In WebSphere, an initial context for a name server is associated with a bootstrap host (machine on which the name server is running) and a bootstrap port. These combined values can be viewed as the address of the name server that owns the initial context. Naming clients can use the JNDI interface or the CosNaming interface to access the name space. Each naming client must be configured specifically to bootstrap (bind) to the name space root of a chosen Application Server, node agent, or deployment manager. In case of J2EE clients, as each

client runs in its own JVM, the bootstrap host and port properties can be set as environment variables of the JVM. In case of CORBA clients, these properties either are specified in a configuration file or can be passed to the server as run-time properties.

A JNDI binding to a name space can be a *simple name* (used when the object being looked up is located on the same Application Server), a *corbaname*, or a *compound name* (fully qualified name). The corbaname form is used when performing a direct URL lookup using a previously obtained initial context.

Figure 2-4 shows the naming topology of WebSphere Application Server V5.

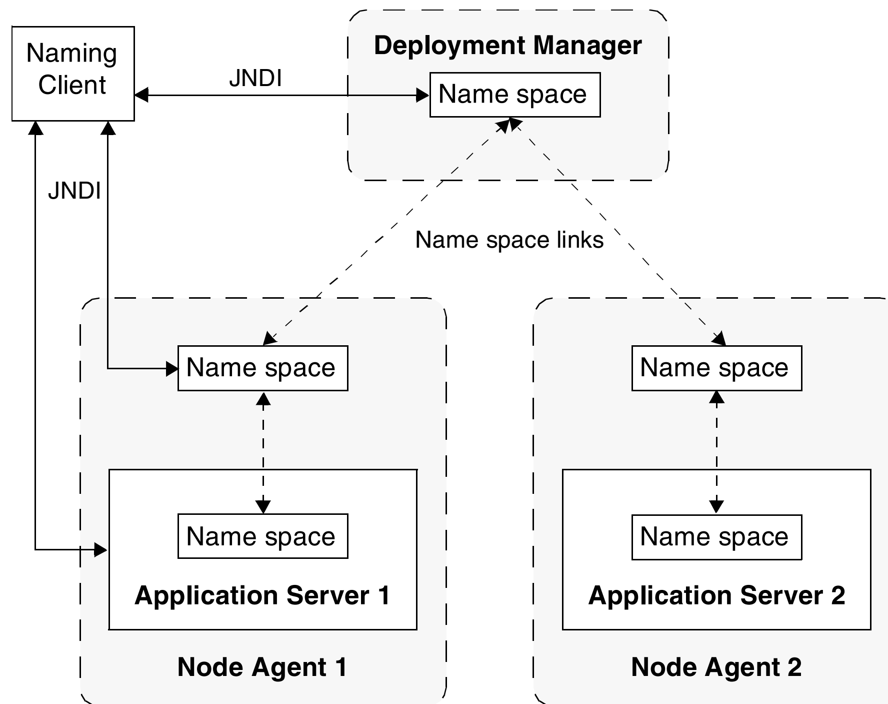


Figure 2-4 Naming topology

Features of WebSphere Application Server V5 name space include:

- The name space is distributed: The name space for a cell is distributed among the various servers. The Deployment Manager, node agent, and Application Server processes all host a name server. System artifacts, such as enterprise bean homes and resources, are bound to the server root of the server they are associated with.
- Transient and persistent partitions: The name space is partitioned into transient areas and persistent areas. Server roots are transient.

System-bound artifacts, such as enterprise bean homes and resources, are bound under server roots. There is a cell persistent root, which can be used for cell-scoped persistent bindings, and a node persistent root, which can be used to bind objects with a node scope.

- Federated name space structure: A name space can contain naming context bindings to contexts located in other servers. If this is the case, the name space is said to be a federated name space, because it is a collection of name spaces from multiple servers. The name spaces link together to cooperatively form a single logical name space.
- Configured bindings: The configuration graphical interface and script interfaces can be used to configure bindings in various root contexts within the name space. These bindings are read-only and are bound by the system at server startup.
- Support for CORBA Interoperable Naming Service (INS) object URLs: WebSphere Application Server V5 contains support for CORBA object URLs (corbaloc and corbaname) as JNDI provider URLs and lookup names.

WebSphere application architecture

One significant difference in the architectures of distributed DCE and WebSphere applications must be considered when migrating DCE applications to WebSphere. As with DCE, the WebSphere infrastructure distinguishes between client and server platforms concerning the middleware configuration. However, a DCE application server or client can run on every node that is configured into a DCE cell, regardless of whether that node is configured as DCE client (hosting just the DCE client runtime) or server (hosting Security and/or CDS server). In a WebSphere environment, on the other hand, Application Servers must be homed on a WebSphere server platform, so in a new WebSphere environment, all migrated DCE Application Servers must be moved to dedicated WebSphere server platforms. This must be considered when planning the system topology of the distributed environment.

This does not apply to a CORBA environment, because CORBA does not use distinct server and client environments.

2.4 Summary

Table 2-1 on page 48 summarizes the information presented in this section. It provides the following information:

- The DCE service that you will need to replace.
- The industry standards related to the service.
- The IBM implementation.

- The major platforms on which the implementation is available. (The list may not be complete.)
- The programming language or languages for the implementation.
- The data that you will need to migrate to the new implementation.

Table 2-1 Summary of recommended replacement technologies

Technology	Services	Industry standards	IBM implemen-tations	Major Platforms	Programming languages	Data migration
aznAPI	PAC Authorization UUID	The Open Group aznAPI	IBM Tivoli Access Manager	AIX Linux Solaris Windows	C/C++ Java	Users, groups, and group memberships
CORBA	Any or all of the RPC services	OMG specifications	IBM WebSphere Application Server	AIX Linux OS/400 OS/390 Solaris Windows	Java C/C++ (client only)	Not supported
DCE RPC	Any or all of the RPC services	DCE	None	Not applicable	Not Applicable	Not applicable
DCE UUID	UUID	DCE	None	Not applicable	Not applicable	Not applicable
J2EE	Authentication Authorization Backing store Delegation Directory Messaging PAC Protection RPC (any or all of the RPC services)	J2EE deployment specification J2EE technology standards IETF standards CORBA standards	IBM WebSphere Application Server	AIX Linux HP-UX OS/400 OS/390 Solaris Windows	Java	Users, groups, and group memberships
Kerberos Version 5	Authentication Delegation GSS-API Login Protection	IETF RFCs: 1510 1964 2743 2744	IBM Network Authentication Service	AIX OS/390 OS/400 (client only)	C	Principals and policies

Technology	Services	Industry standards	IBM implemen-tations	Major Platforms	Programming languages	Data migration
LDAP	Directory Registry Backing store ERA	IETF RFCs: 1274 1777 1778 1779 1823 2052 2219 2222 2247 2251-2256 2891	IBM Directory Server	AIX HP-UX Linux RedHat Solaris Windows	C Java (using JNDI interfaces)	Registry and ERA data
NTP	Time	IETF RFC 1305	IBM AIX (Version 4.2 or later)	AIX	C	Not applicable
Platform auditing	Auditing	None	IBM AIX	AIX	C/C++	Not applicable
Platform logging and messaging/ UNIX logging and messaging	Serviceability	The Open Group UNIX 98, POSIX 1-1996 standards	All IBM operating systems	All IBM platforms	C/C++	Message catalog files
Platform logging and messaging/ Windows logging and messaging	Serviceability	None	None	Not applicable	Not applicable	Not applicable
POSIX 1003.1c	Threads	POSIX 1003.1c	IBM AIX	AIX	C	Not applicable
Web services (for C/C++)	Any or all of the RPC services	WSDL SOAP UDDI	Web services toolkit WebSphere Application Server	Not known	Java[1]	Not supported

[1] IBM supports a Java implementation of Web services in IBM WebSphere Application Server.

3

Replacement strategies

This chapter recommends strategies for replacing direct dependencies on DCE services:

- The first part recommends strategies for applications written in C and C++.
- The second part recommends strategies for applications migrated to Java.
- The third part recommends strategies for applications in which a combination of C/C++ and Java is used.

This chapter references technologies and IBM implementations of these technologies. For information about these technologies and IBM implementations, see Chapter 2, “Replacement technologies” on page 19.

© Copyright IBM Corp. 2003. All rights reserved.

3.1 Replacement strategies for C/C++ applications

The following subsections recommend C/C++ strategies for replacing direct dependencies on many DCE services. Each section presents a DCE service, states whether a C/C++ strategy is recommended to replace direct dependencies to the service, and provides a high-level description of implementing the strategy.

3.1.1 Auditing

The recommended C/C++ strategy for replacing direct dependencies to the DCE auditing service is to use auditing interfaces provided by the operating system platform. (See section 2.2.8, “Platform auditing” on page 31.)

Take the following steps to implement this strategy.

Administrator

The administrator must use platform auditing interfaces rather than DCE auditing interfaces.

Developer

The developer must revise the applications to use platform API auditing interfaces rather than DCE API auditing interfaces.

Many application environments indirectly depend on configuring some of the DCE servers to audit internal events on the server. If your environment has this indirect dependency, you must consider whether the replacement servers that you will use offer similar auditing capability.

Table 3-1 lists each DCE server that can be configured for auditing, the server that is recommended for C/C++ applications to use as a replacement for this DCE server, and whether the recommended replacement server can be configured for auditing.

Table 3-1 DCE server replacements for auditing

DCE server	Recommended replacement server	Auditing?
DCE authentication server	IBM Network Authentication Service server	See the following note
DCE directory server	IBM Directory Server	Yes
DCE privilege server	IBM Tivoli Access Manager server	Yes
DCE registry server	IBM Directory Server	Yes
DCE time server	IBM AIX NTP server	No

> **Note:** The IBM Network Authentication Service servers cannot be configured explicitly for auditing. However, the IBM Network Authentication Service KDC server can be configured to log information about all of the attempts to get initial and service tickets. In addition, the administration server of the IBM Network Authentication Service can be configured to log information about the attempts to store authentication data. The log information might be used as an audit trail.
>
> Also, in a configuration where IBM Network Authentication Service uses an IBM Directory Server as its user registry, the IBM Directory Server auditing services can be used.

3.1.2 Authentication

No DCE applications depend directly on the DCE authentication service because this service can be accessed indirectly only by means of the DCE GSS-API or RPC security services. For applications that indirectly depend on the DCE authentication service by means of DCE GSS-API, see 3.1.6, "Delegation, GSS-API, and login" on page 56. For applications that indirectly access the DCE authentication service by means of DCE RPC security, see 3.1.19, "RPC services" on page 63.

3.1.3 Authorization, PAC, and UUID

The following alternate C/C++ strategies are recommended for replacing direct dependencies on the DCE authorization and PAC services:

Option 1 Use IBM Tivoli Access Manager and IBM Directory Server. IBM Tivoli Access Manager contains the IBM implementation of the aznAPI technology. (See section 2.2.1, "aznAPI" on page 22 for more information.) IBM Directory Server is the IBM implementation of the LDAP technology. (See section 2.2.6, "LDAP" on page 28 for more information.)

Option 2 Use user-written code, IBM Directory Server, and DCE UUID code. (See section 2.2.4, "DCE UUID" on page 26 for more information.)

Option 1: IBM Tivoli Access Manager

This strategy is recommended for customers who do not want to develop and maintain their own authorization code. Using this option, an external product (IBM Tivoli Access Manager) handles all of the logic associated with authorization, including storing ACLs and objects in a database, getting PACs for users, and determining authorization of users. IBM Tivoli Access Manager is

configured to use IBM Directory Server as its LDAP-based database for group information so that DCE data can be migrated to IBM Tivoli Access Manager.

The following steps are required to implement this strategy.

Administrator

The administrator must make the following changes to the application environment:

- Install the IBM Directory Server.
- Install the IBM Tivoli Access Manager servers.
- Configure privilege data (users, groups, and group memberships) in the IBM Tivoli Access Manager privilege database. Some of this data can be migrated from the DCE registry database by performing this procedure:
  a. Load the IBM Tivoli Access Manager schema into IBM Directory Server.
  b. Upgrade to DCE Version 3.2 and use DCE Version 3.2 to migrate the entire DCE registry database to IBM Directory Server. (Prior versions of DCE do not support this registry migration.)
  c. Configure the IBM Tivoli Access Manager privilege database so that it uses some of the DCE-migrated data. Note that this procedure must be done in a certain sequence. Refer to Chapter 5, “Using DCE objects with IBM Tivoli Access Manager” on page 85 and to the document *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.
- Configure the authorization data (object and ACL data), which currently is in an application-specific database, into the IBM Tivoli Access Manager authorization database. IBM does not provide a procedure for migrating this data from the application-specific authorization database into the IBM Tivoli Access Manager authorization database.
- Manage the privilege and authorization data using the interfaces of the IBM Tivoli Access Manager rather than DCE. (Exception: DCE and IBM Tivoli Access Manager can be configured to share some of the same privilege data, which might be helpful during a staged migration. If you are sharing privilege data, you must manage this data using a combination of DCE and Tivoli Access Manager interfaces. Refer to Chapter 5, “Using DCE objects with IBM Tivoli Access Manager” on page 85.

Developer

The developer must make the following changes to the applications:

- Remove any user-written authorization code, back-end *rdacl* code, and user-written ACL database code.

- Remove any calls to the DCE authorization APIs (dce_acl_*, priv_*, sec_acl_*, sec_cred_*, sec_id_*, rpc_binding_*auth_*)
- Revise the application servers to control access to objects by calling the IBM Tivoli Access Manager API interfaces (azn*). The IBM Tivoli Access Manager API accept, as input, an identity (such as a Kerberos identity), an object, and the desired permissions. Then, IBM Tivoli Access Manager outputs an authorization decision. The logic for acquiring and reading a user PAC, finding an object, finding and reading the ACL on the object, and making a comparison between the PAC, the ACL, and the desired permissions is handled internally by IBM Tivoli Access Manager.

Option 2: User-written authorization code

This strategy is recommended for customers who have a large investment in user-written authorization code and want to continue using this code. This strategy also is recommended for customers who want to have complete control over their authorization code.

The following steps are required to implement this strategy.

Administrator

The administrator must make the following changes to the application environment:

- Install the IBM Directory Server.
- Acquire the DCE UUID code, and use it to create a DCE UUID generator. The DCE UUID code can be acquired, for example, from *The Open Group* (http://www.opengroup.org).
- Upgrade to DCE Version 3.2, and migrate the entire DCE registry database to the IBM Directory Server. (Refer to the document *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide.*) Note that prior versions of DCE do not support this migration.
- Manage the privilege data (user, group, and group membership data) using the interfaces of IBM Directory Server and the DCE UUID generator. To determine where migrated privilege data is stored, refer to the information on the DCE LDAP schema as documented in the *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.

Developer

The developer must make the following changes to the applications:

- If the backend object and ACL databases were created using the DCE backing store APIs (dce_db_*), move these databases to the replacement strategy you are using for the DCE backing store service.

- Replace all of the calls to the DCE authorization APIs (dce_acl_*, priv_*, sec_acl_*, sec_cred_*, sec_id_*, rpc_binding_*auth_*) with user-written code.
- Assuming that the replacement authentication service is IBM Network Authentication Service or another implementation of the Kerberos technology, the application server can receive only a principal name from this service. Therefore, add the following logic to the application server:
  - Using a principal name, search IBM Directory Server for the UUID of the principal and the UUID of each group to which the principal is a member. To determine where in IBM Directory Server this data is stored, refer to the information on the DCE LDAP schema in the document *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.
  - Pass this information (the same information that is in a PAC) to the user-written authorization code.

3.1.4 Backing store

The recommended C/C++ strategy for replacing direct dependencies to the DCE backing store service is to use IBM Directory Server, which is the IBM implementation of the LDAP technology. (See section 2.2.6, “LDAP” on page 28 for more information.)

The following steps are required to implement this strategy.

Administrator

The administrator must install the IBM Directory Server and manually configure, in IBM Directory Server, the data that currently is configured in the DCE back-end database.

Developer

The developer must revise the applications to access back-end data using the LDAP-APIs of IBM Directory Server rather than DCE.

3.1.5 Configuration

There is no recommended strategy for replacing the DCE configuration service. This service configures DCE and is not needed when DCE is removed.

3.1.6 Delegation, GSS-API, and login

The recommended C/C++ strategy for replacing direct dependencies to the DCE delegation, GSS-API, and the login services is to use IBM Network

Authentication Service, which is the IBM implementation of the Kerberos technology. (See section 2.2.5, “Kerberos” on page 26 for more information.)

The following steps are required to implement this strategy.

Administrator

The administrator must make the following changes to the application environment:

- Install the IBM Directory Server.
- Install the IBM Network Authentication Service servers.
- Configure authentication data in the authentication database of IBM Network Authentication Service. This data can be migrated from the DCE registry database with the following procedure:
  a. Upgrade to DCE Version 3.2 and use DCE Version 3.2 to migrate the entire DCE registry database to the IBM Directory Server. (Refer to the document *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.) Note that prior versions of DCE do not support this migration.
  b. Configure IBM Network Authentication Service to use DCE-migrated data. (Refer to Chapter 4, “Using DCE data with IBM Network Authentication Service” on page 75 and to the document *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.)
- Manage the authentication data using the interfaces of IBM Network Authentication Service rather than DCE. (Exception: DCE and IBM Network Authentication Service can be configured to share some of the same authentication data, which might be helpful during a staged migration. If you are sharing this data, you must manage it using a combination of DCE and IBM Network Authentication Service interfaces. Refer to Chapter 4, “Using DCE data with IBM Network Authentication Service” on page 75.)

Developer

The developer must make the following changes to the applications:

- Revise any DCE login and delegation API calls (sec_login_*) to use the Kerberos APIs provided with IBM Network Authentication Service. If the application makes use of sealed delegation, this must be changed to impersonation delegation because the Kerberos technology only supports impersonation delegation.
- Revise any gss_init_sec_context calls to use Kerberos rather than the DCE GSS-API mechanism.
- Remove any DCE-specific GSS-API calls (gssdce_*) and, if required, replace them with comparable API calls.

Note: At the time of writing, IBM does not offer Network Authentication Service on the Windows platform. However, Windows 2000 can be configured to be a Kerberos client to an IBM Network Authentication Service server, as demonstrated in Chapter 8, “Scenario 1: GSS-API application” on page 125.

3.1.7 Directory

The recommended C/C++ strategy for replacing direct dependencies on the DCE directory service is to use IBM Directory Server, which is the IBM implementation of the LDAP technology. (See section 2.2.6, “LDAP” on page 28 for more information.)

The following actions are required to implement this strategy.

Administrator

The administrator must install IBM Directory Server and configure in it the data that the application needs to access directly.

Developer

The developer must revise the applications to access directory data using the APIs of IBM Directory Server (which are LDAP APIs) rather than the DCE APIs.

3.1.8 Extended Registry Attributes

The recommended C/C++ strategy for replacing the ERA service is to use IBM Directory Server, which is the IBM implementation of the LDAP technology.

The following steps are required to implement this strategy.

Administrator

The administrator must make the following changes to the application environment:

- Install the IBM Directory Server.
- Upgrade to DCE Version 3.2 and use it to migrate the entire DCE registry database to IBM Directory Server. (Refer to the document *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.) Prior versions of DCE do not support this registry data migration to an LDAP directory.
- Manage the ERA data using the interfaces of IBM Directory Server rather than DCE. To determine where in IBM Directory Server the migrated ERA data is stored, refer to the information on the DCE LDAP schema as documented in *IBM DCE Version 3.2 for AIX and Solaris: DCE Security*

Registry and LDAP Integration Guide. As noted in this documentation, the ERA data is stored as part of a binary structure, so it will be necessary to parse this structure to obtain the ERA data. To determine how to read the binary ERA data, read Chapter 6, "Binary structure of DCE ERA data in LDAP" on page 105. To create new ERA types, use the interfaces of IBM Directory Server to add new attribute types to the IBM Directory Server schema.

Developer

The developer must change the application to use the APIs of IBM Directory Server (which are LDAP APIs) rather than DCE APIs to read ERA data.

3.1.9 Event management

There is no C/C++ recommended strategy for replacing direct dependencies to the DCE event management service. This service monitors events on DCE servers. When DCE is removed, this service is no longer needed.

Your application might indirectly depend on monitoring events on DCE servers. If this is the case, check each implementation that you will use to replace the DCE servers to determine whether the replacement implementation offers a similar event monitoring capability.

3.1.10 GSS-API

See section 3.1.6, "Delegation, GSS-API, and login" on page 56.

3.1.11 Host management

There is no C/C++ recommended strategy for replacing direct dependencies to the DCE host management service. Few DCE applications directly depend on this service.

3.1.12 Integrated login

There is no C/C++ recommended strategy for replacing the DCE integrated login service because this service cannot be accessed directly by DCE applications. Many application environments indirectly depend on users using their local platform to perform a login that is integrated with the DCE login service. If your application has this indirect dependency, you need to consider whether your local operating system platform can perform an integrated login with the replacement strategy you are using for the DCE login service.

IBM AIX supports integrated login with IBM Network Authentication Service. Many other platforms also support integrated login. For example, many UNIX platforms support Pluggable Authentication Modules (PAM), and all Windows platforms support Graphical Identification and Authentication (GINA).

3.1.13 Login

See section 3.1.6, “Delegation, GSS-API, and login” on page 56.

3.1.14 Messaging

The recommended C/C++ strategy for replacing direct dependencies on the DCE messaging service is to use the platform messaging interfaces. (See section 2.2.9, “Platform logging and messaging” on page 31 for more information.) Note that messaging in this context refers only to the handling of messages within an application and not to the communication of messages between applications, which is handled by a different type of service. Because UNIX and Windows messaging are both designed to be used by a single system, neither supports the concept of associating a unique cell-wide number, such as a status code, with a message.

To implement this strategy using UNIX messaging, the developer must refer to the platform-specific documentation on how to use XPG4 messaging to support multiple languages. On the AIX platform, see Chapter 16, National Language Support, of the *AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs*, available from the AIX library at:

http://publibn.boulder.ibm.com/cgi-bin/ds_form?lang=en_US&viewset=AIX

Similar documentation for Solaris can be found in Chapter 7, Writing Internationalized Code, of the *Solaris Internationalization Guide For Developers*, available at the following Web site:

http://docs.sun.com/db/doc/802-5878?q=catopen

Because UNIX messaging uses XPG4 messaging standards, which are the standards that DCE uses, you can use the DCE message source files (*.msg) and message output files (*.cat) without making any changes to these files.

To implement this strategy using Windows message handling, the developer must refer to the Windows documentation on how to use Windows message resource files and functions to support multiple languages based on the Windows regional settings. For a discussion and examples of Microsoft

proprietary messaging support, see the document *Writing Multilingual User Interface Applications*, available at:

http://www.microsoft.com/globaldev/handson/dev/muiapp.mspx

Because Windows messaging files are in a proprietary format, migrate the DCE message source files to the Windows format.

3.1.15 PAC

See section 3.1.3, “Authorization, PAC, and UUID” on page 53.

3.1.16 Password strength

There is no C/C++ recommended strategy for replacing direct dependencies to the DCE password strength service. Few applications directly depend on this service.

Many applications indirectly depend on being able to configure DCE to enforce a password strength policy when processing password create and password change commands. If you have this indirect dependency, consider whether you still will be using passwords with your replacement strategy for the DCE login service and, if so, whether the replacement strategy you are using to create and change these passwords offers a password strength service.

If you plan to use the strategies recommended in this paper for C/C++ applications, you will use the login service provided by IBM Network Authentication Service and the administrative interfaces of IBM Network Authentication Service to create and update passwords for the login service. The administrative interfaces of IBM Network Authentication Service can be configured to enforce the following password strength policies:

- Password dictionary file
- Password maximum lifetime
- Password minimum lifetime
- Password length

If you need more powerful password strength enforcement, do the following:

1. Develop an interface for users to use in creating and modifying passwords. The interface needs to call a password strength server to validate the strength of the password or to generate a password. Then, make the appropriate call to the Network Authentication Service interfaces for creating and changing passwords.

2. Acquire or develop a password strength server. The password strength server needs to perform expanded password checking based on the configuration of password rules, dictionary lists, and policies.

3.1.17 Protection

There are no DCE applications that directly access this service because it can be accessed indirectly by means of the DCE GSS-API or RPC security services only. For applications that indirectly access the DCE protection service by means of DCE GSS-API, see section 3.1.6, "Delegation, GSS-API, and login" on page 56. For applications that indirectly access the DCE protection service by means of DCE RPC security, see section 3.1.19, "RPC services" on page 63.

3.1.18 Registry

The recommended C/C++ strategy for replacing direct dependencies to the DCE registry service is to use IBM Directory Server, which is the IBM implementation of the LDAP technology. (See section 2.2.6, "LDAP" on page 28 for more information.)

The following steps are required to implement this strategy.

Administrator

The administrator must make the following changes to the application environment:

- Install the IBM Directory Server.
- Upgrade to DCE Version 3.2 and use it to migrate the entire DCE registry database to IBM Directory Server. Prior versions of DCE do not support this registry data migration to LDAP. (Refer to the document *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.)
- Manage the registry data using the interfaces of IBM Directory Server rather than DCE. To determine where in IBM Directory Server the migrated registry data is stored, refer to the information on the DCE LDAP schema as documented in *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.

Developer

The developer must change the application to use the API interfaces of IBM Directory Server rather than DCE to read registry data.

3.1.19 RPC services

The following alternate strategies are recommended for replacing direct dependencies to the DCE RPC services:

Option 1 Use an implementation of the CORBA technology. (See section 2.2.2, “CORBA” on page 23 for more information.)

Option 2 Use an implementation of the Web Services technology. (See section 2.2.11, “Web services” on page 33 for more information.)

Option 3 Use a stand-alone implementation of the DCE RPC technology.

Option 1: Implementation of CORBA

This strategy uses a C/C++ implementation of the CORBA technology to replace all of the DCE RPC services. Because different implementations of CORBA are not always able to interoperate, it is recommended that you either use the same CORBA implementation for the application clients and servers, or use different CORBA implementations that have been tested against each other.

The following steps are required to implement this strategy.

Administrator

The administrator must make the following changes to the application environment:

- Install the servers provided by the CORBA implementation. For example, it might be necessary to install a security server provided by the CORBA implementation.
- If the CORBA implementation you are using provides security support for clients and servers, configure the authentication data required by the CORBA implementation. For example, if the CORBA implementation performs authentication based on user identities and passwords, user identities and passwords must be configured for all user clients. If the CORBA implementation performs authentication based on public certificates, it will be necessary to acquire certificates for all user clients and configure a way of mapping these certificates to user identities.

Developer

The developer must revise, redesign, and re-code the application to use CORBA rather than the DCE RPC programming model.

Option 2: Web services

This strategy uses a C/C++ implementation of the Web services. As Web services is a new technology, the following must be considered:

- Few implementations of Web services currently exist for a C/C++ development environment.
- At the time of writing, a standard for Web services security has not been set. For information about the proposed standard, refer to:

  `http://www.ibm.com/developerworks/library/ws-secure/`

The following steps are required to implement this strategy.

Administrator

The administrator must make the following changes to the application environment:

- Install the servers provided by the Web Services implementation. For example, it might be necessary to install a security server provided by the Web Services implementation.
- Configure the authentication data required by the Web Services implementation. For example, if the Web Services implementation performs authentication based on user identities and passwords, user identities and passwords must be configured for all user clients. If the Web Services implementation performs authentication based on public certificates, it will be necessary to acquire certificates for all user clients and configure a way of mapping these certificates to user identities.

Developer

The developer must redesign the application to use the Web services rather than the DCE RPC programming model and migrate the DCE RPC data to the data format required by Web services. The following provides a loose mapping between DCE RPC functions and the Web services counterparts:

- The DCE application client and server map to the Web services service requestor and provider.
- The DCE RPC export and import operations map to the Web services publish and find operations.
- The DCE RPC bind operation maps to the Web services bind operation.
- The DCE RPC manager routine invoked by the DCE application server maps to the actual Web services implementation.
- The DCE RPC namespace maps to a subset of the Web services service registry. (The information stored in the service registry is more extensive than

the information stored in the DCE RPC namespace, as noted in the next bullet.)

- The DCE RPC IDL specification maps to the Web services service description. The mapped information includes data types, operations, binding information, and network location. However, with Web services, the service requester can discover all of this information dynamically during run time. With DCE, the application client statically links in the data type and operation information and discovers only the binding and network location information dynamically during run time.

Option 3: A stand-alone offering of DCE RPC

> **Note:** This optional strategy is not available currently because a stand-alone offering of DCE RPC does not exist as a product.

This optional strategy is applicable only if a stand-alone offering of the DCE RPC technology becomes available. This offering would have to include, at minimum, an implementation of the RPC basics service, which is comprised of:

- IDL compiler (which generates client/server stub code)
- RPC run-time server routines (rpc_server_*)
- RPC run-time binding routines (rpc_binding_*)
- RPC run-time management routines (rpc_mgmt_*)
- RPC run-time object routines (rpc_object_*)
- RPC run-time memory management routines (rpc_sm_*, rpc_ss_*)
- UUID routines (uuid_*)

It also might include:

- RPC namespace APIs (rpc_ns_*). These APIs would use a non-DCE directory server, such as an LDAP directory server, for storing namespace information.
- RPC endpoint APIs (rpc_ep_* and rpc_mgmt_ep_*). These APIs would use a non-DCE host or file server, such as a stand-alone implementation of the DCE host server or a non-DCE distributed file server, for storing endpoint information.
- RPC security (rpc_*auth*) APIs. These APIs would use a non-DCE security server, such as the Kerberos KDC server provided by IBM Network Authentication Service, to process authentication and protection protocols.

If such an offering were made available and required the use of non-DCE servers, the administrator would have to migrate to the non-DCE servers. However, the developer probably would have to make only minor modifications to

the application, if any, because it would be expected that the RPC API interfaces would remain the same.

3.1.20 Serviceability

There is no recommended C/C++ strategy for replacing direct dependencies to all of the functions offered by DCE serviceability. However, a platform-specific strategy is recommended to replace the primary function offered by DCE serviceability, which is the logging function. On UNIX platforms, the recommended strategy is to use UNIX logging. On Windows platforms, the recommended strategy is to use Windows logging. As stated previously, these two strategies replace only the logging function and do not replace other serviceability functions, such as controlling the types of messages that are logged and forwarding messages to an event management service.

To implement this strategy on UNIX-compliant platforms, the developer must refer to the platform-specific documentation on the `syslog()` function. This function is somewhat similar to DCE serviceability. This error-logging scheme does not automatically fetch a status code's error text from a message catalog as DCE serviceability does, but it supports a similar type of facility-based logging. For a discussion and examples of how to use the `syslog()` interface on the AIX platform, see the *AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions, Volume 2*, available at the following Web site:

http://publibn.boulder.ibm.com/cgi-bin/ds_form?lang=en_US&viewset=AIX

Similar documentation for Solaris can be found in the online Sun Solaris product documentation, section 3: *Basic Library Functions, syslog(3C)* at:

http://docs.sun.com/db/doc/806-0627/6j9vhfn8g?a=view

To implement this strategy on the Windows platform, the developer must refer to the Windows documentation describing Windows proprietary logging to the Windows Event Log, using functions such as `RegisterEventSource()` and `ReportEvent()`. When used in conjunction with message resource files, Windows can log message text that is translated into the language that matches the Windows regional settings. Search for *Event Logging Operations* for an overview of Windows event logging and the associated API at:

http://msdn.microsoft.com/

Many application environments indirectly depend on configuring DCE servers to log internal events on the server. If this is the case, check each replacement server to determine whether the replacement server offers a similar logging capability.

3.1.21 Threads

The recommended C/C++ strategy for replacing the DCE threads service is to use an implementation of the POSIX 1003.c draft 10 threads technology. An IBM implementation of this technology is available with AIX. Some independent software vendors offer implementations of POSIX 1003.c draft 10 on other operating system platforms. (For further information, see also section 2.2.10, "POSIX 1003.1c threads" on page 32.)

To implement this strategy, the developer must port the DCE pthread API calls (which are based on draft 4 of POSIX 1003.c) to the pthread calls defined by draft 10 of POSIX 1003.c. The following provides a high-level description of the types of porting issues that a developer must address.

Non-portable extensions in the draft 4 pthreads API

Draft 4 pthreads defines non-portable API routines that do not have equivalent draft 10 pthreads API routines. These routines are designated by their `_np` extensions.

Changes in API syntax

A number of pthread API routines must be mapped to a new syntax. As an example, the draft 4 pthread routine `pthread_mutexattr_create` has been renamed to `pthread_mutexattr_init` in draft 10.

Changes in data types

Some draft 4 pthreads data types must be replaced with those specified in draft 10. For example, the draft 4 `pthread_startroutine_t start_routine` pthreads data type must be replaced by `void *(*start_routine) (void *)`in draft 10.

Changes in error codes

Draft 4 pthread functions return a –1 on error and set the global *errno* variable to the appropriate value. Draft 10 pthread functions return all of the error numbers as return values. Applications must be modified to handle error returns correctly. In addition, some draft 10 pthread routines return different *errno* values than draft 4 pthreads do.

Changes in exception handling

Draft 10 pthreads do not support the DCE exception handling functionality. Programs written to handle DCE exceptions must be rewritten to work with draft 10 pthreads. For cancellation clean-up handling, replace TRY/CATCH with `pthread_cleanup_push` and `pthread_cleanup_pop`. Replace all other exception handling with code that handles error returns from the draft 10 `pthread_*` calls.

Changes in thread functions

Some of the basic pthreads functionality has changed in draft 10. This means that either the use of a pthread routine must be altered to preserve the intended functionality or the program must adapt to the new functionality of the pthreads routine. The areas where function has changed are listed next.

Default thread attributes

Draft 10 pthreads use NULL to indicate the use of default attribute properties, instead of `pthread_attr_default`.

Thread detach/join

Draft 10 pthreads create a thread in a joinable state by default, as does draft 4. However, draft 10 allows you to create a thread in a detached state. When you do this, you do not have to call `pthread_detach` to detach the thread.

Scheduling

Significant changes were made in getting and setting thread scheduling policy and priority. For example, draft 10 pthreads require you to use the `sched_param` structure in conjunction with the `pthread_setschedparam` and `pthread_attr_setprio` routines to specify scheduling policy and priority. In contrast, draft 4 uses the `pthread_setprio` routine.

Mutexes

Draft 4 pthreads support three types of mutexes: fast, non-recursive, and recursive. Draft 10 pthreads support non-recursive mutexes only. Applications requiring other mutex types must provide them themselves.

Condition variables

Draft 4 pthreads provide a default value of `pthread_condattr_default`; in draft 10, this value is specified with a NULL value. Draft 10 supports static initialization of condition variables and has changed the arguments to condition variable initialization.

Thread cancellation

Draft 10 pthreads change the syntax of the pthread routines pertaining to thread cancellation. In addition, the routines now return the previous state as a function parameter.

Pre- and post-fork handling

The order of invocation of pre-fork handlers has been changed to last in, first out (LIFO) in draft 10 pthreads. The order of post-fork handlers is first in, first out (FIFO).

Changes in reentrant C library function

There are syntactical differences in the DCE libc_r and AIX libc_r routines. However, the basic functionality of the routines is the same and just needs to be mapped.

3.1.22 Time

There is no recommended C/C++ strategy for replacing direct dependencies to the DCE time service. It is assumed that applications do not depend directly on this service.

Many applications indirectly depend on using a time service to synchronize the time on each system containing a DCE security authentication server, as this server is time-sensitive. For applications that currently use the DCE time service to handle this indirect dependency, the recommended replacement is to use a platform-specific implementation of the Network Time Protocol (NTP). AIX includes an *xntpd* daemon, which is an AIX implementation of NTP. Implementations of NTP are available on other platforms and some of these implementations are freely available. For information about NTP, refer :

http://www.ntp.org/index.html

3.1.23 UUID

See section 3.1.3, “Authorization, PAC, and UUID” on page 53 for more information.

3.2 Replacement strategy for Java applications

This section recommends a Java strategy for replacing direct dependencies to DCE services. The strategy is to use IBM WebSphere Application Server and IBM Directory Server. IBM WebSphere Application Server is the IBM implementation of the J2EE technology. (See section 2.3, “Technologies for Java applications” on page 34 for more information.) The IBM Directory Server is the IBM implementation of the LDAP technology. (See 2.2.6, “LDAP” on page 28 for more information.)

To implement this strategy:

- The architect must determine a new J2EE component architecture for the DCE applications.
- The administrator must revise the application environment for the new architecture.

- The developer must rewrite the DCE applications to run in the new environment.

3.2.1 Determining a new architecture

The architect must determine a new J2EE component architecture to which each DCE application will be rewritten. The following presents two examples of J2EE component architectures. For complete information about J2EE component architectures, refer to the J2EE and IBM WebSphere Application Server documentation.

Example 1: Java client applications and enterprise beans

This example maps to a DCE application consisting of application clients, application servers, and back-end application servers, with all communications being performed using the DCE RPC protocol.

In this example, the DCE RPC clients are rewritten to be Java client applications, the DCE RPC servers are rewritten to be enterprise beans, and the back-end DCE RPC servers are rewritten to be back-end enterprise beans.

Figure 3-1 illustrates this example:

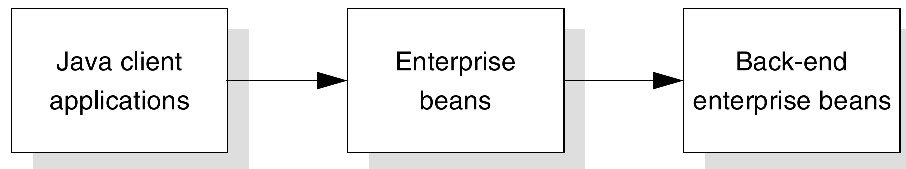


Figure 3-1 Java client and back-end enterprise beans

The Java client application or the client enterprise bean finds the desired method using the Java Naming and Directory Interface (JNDI) and invokes the desired method using Java Remote Method Invocation (RMI). If the enterprise bean containing the desired method is located on a remote node, the client serializes, marshals, and transmits the method invocation using CORBA IIOP and, if necessary, secures the IIOP transmission using CORBA CSIv2.

The administrator can configure CSIv2 so that client identity information is passed to the target enterprise bean using client basic authentication (client passes its user identity and password), client certificate authentication (client passes its public certificate using an SSL transport), delegation, or identity assertion. If CSIv2 is configured to use an SSL transport, SSL passes the public certificate of the target enterprise bean to the client and protects the privacy and integrity of all transmitted data.

Example 2: Web clients, servlets, and enterprise beans

This example maps to a DCE application consisting of application clients, application servers, and back-end application servers. In the DCE application, communications between the application clients and servers is performed with a Web protocol, and communications between the application servers and back-end application servers is performed using the DCE RPC protocol.

In this replacement example, the application clients are rewritten to be Web clients (for example, browsers, Java applets, or Java client applications), the application servers are rewritten to be servlets, and the back-end application servers are rewritten to be enterprise beans.

Figure 3-2 illustrates this example:

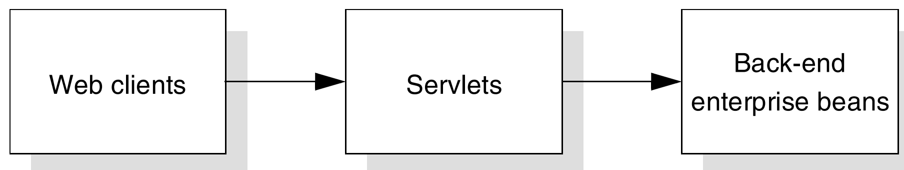


Figure 3-2 Web clients, servlets, and back-end enterprise beans

The Web client finds the servlets using some external method (such as an Internet search engine), uses HTTP request and response commands to communicate with the servlet, and if necessary, secures communications using a Web login method.

The Web login method determines the authentication information that the client passes to the servlet. The information might be HTTP basic authentication (client passes its user identity and password), HTTP form-based authentication (client passes its user identity and password in a HTML form), or HTTPS (client passes its public certificate using an SSL transport). If SSL is configured, which it is with HTTPS, SSL passes the public certificate of the servlet to the client and protects the privacy and integrity of all transmitted data.

The servlets invoke methods on enterprise beans using the set of protocols described in “Example 1: Java client applications and enterprise beans” on page 70 (JNDI, RMI, IIOP, and CSIv2). This makes it possible to delegate or assert a client identity to downstream enterprise beans.

3.2.2 Revising the application environment for the new architecture

The administrator must revise the application environment required to run applications in the new architecture by doing the following:

- Install WebSphere Application Server servers.
- If IBM Directory Server is being used as the registry database:
  - Install IBM Directory Server.
  - Configure each IBM WebSphere Application Server to use IBM Directory Server as its registry database.
  - Configure users and groups in IBM Directory Server. This information can be migrated from DCE by upgrading to DCE Version 3.2, using it to migrate the entire DCE registry database to IBM Directory Server, and configuring IBM WebSphere Application Server to use DCE attributes for user identity, group identity, and group membership information. For more information about moving the DCE registry to the IBM Directory Server, see the *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.

    For an example of configuring WebSphere Application Server to use the DCE attributes that have been migrated to the IBM Directory Server, see Chapter 10, “Scenario 3: Secure RPC application #1” on page 195.
  - If authentication will be based on user passwords, configure a password in IBM Directory Server for each user. (DCE passwords cannot be migrated to IBM WebSphere Application Server.) If authentication will be based on certificates, obtain a certificate for each user and configure a mapping between the certificate and a user stored in IBM Directory Server. (IBM WebSphere Application Server provides administrative tools to do this mapping).

    For an example of configuring a mapping between certificates and migrated DCE identities, see “Configure LDAP filter rules” on page 218.
  - Configure each IBM WebSphere Application Server to use Lightweight Third Party Authentication (LTPA) as its authentication mechanism. Using LTPA, IBM WebSphere Application Server will use information in its registry database (in this case, IBM Directory Server) to validate authentication information received from the client and associate a credential, which is similar to a DCE PAC, with the client context.
  - Configure any or all of the following as required by the J2EE architecture that the applications will be using: CSIv2, SSL, and a Web login method.

3.2.3 Rewriting the DCE applications to the new architecture

To port a DCE application component to J2EE components, the developer must:

- Remove from the DCE application all of the system processing that will be handled by the container
- Replace the remaining code with Java APIs
- Compile and package the application as J2EE components
- Configure the container for each component

3.3 Replacement strategies for mixed applications

This section recommends mixed strategies for replacing direct dependencies to DCE services. Each strategy uses a mixture of the C/C++ and Java strategies described in the previous two sections. These strategies might be useful for customers who want to migrate to Java but wish to preserve some C/C++ code.

3.3.1 CORBA interoperability

This strategy makes use of the CORBA C/C++ client libraries and the enterprise bean CORBA interoperability provided with IBM WebSphere Application Server. Using these libraries and the interoperability, it is possible to do the following:

- Rewrite a DCE application server as an enterprise bean and configure the enterprise bean so that it interoperates with CORBA application clients.
- Revise the C/C++ DCE application client so that it communicates with the enterprise bean, which is written in Java, using CORBA protocols.

The strategy has a few limitations, which are as follows:

- The CSIv2 provided by IBM WebSphere Application Server for the C/C++ libraries supports only one method of passing client identity information to the target enterprise bean. The supported method is client certificate authentication (client passes its public certificate using an SSL transport).
- The data types used by the interfaces in the enterprise bean must be data types defined in either the CORBA Java API mapping specification or in the value type libraries provided by IBM WebSphere Application Server.

3.3.2 Java Native Interface

This strategy makes use of the Java Native Interface (JNI) APIs provided by IBM WebSphere Application Server in Java 2 Standard Edition (J2SE). Using JNI, it is

possible to wrap a C/C++ application as a Java applet, Java client application, or servlet.

Note: It also is possible to wrap a C/C++ application in an enterprise bean. However, the J2EE specifications state that an enterprise bean should not use JNI. For this reason, this paper does not recommend wrapping a C/C++ application in an enterprise bean, but instead recommends using JCA in combination with JNI as described in the following section.

3.3.3 JCA and JNI

This strategy makes use of the Java Connector Architecture (JCA) provided by IBM WebSphere Application Server along with JNI. Using JCA, it is possible to connect an enterprise bean, servlet, or Java client application to a simple, local resource adapter. Using JNI, it is possible to interface the resource adapter to a local C/C++ application.

Figure 3-3 illustrates an enterprise bean, a resource adapter, and a C/C++ application. In this example, the enterprise bean and resource adapter serve as wrappers for the C/C++ application.

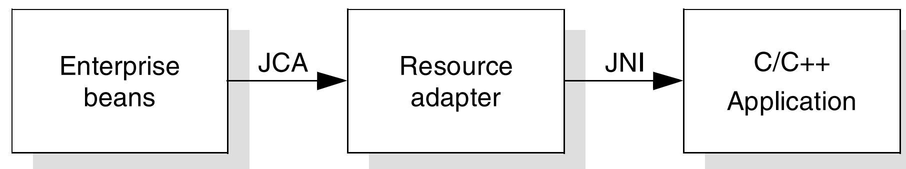


Figure 3-3 JCA and JNI

For an example of using JCA and JNI, see “Scenario 3: Secure RPC application #1” on page 195.

4

Using DCE data with IBM Network Authentication Service

This chapter is for administrators who want to use DCE authentication data with IBM Network Authentication Service, which basically provides Kerberos functionality as a means for authentication. In order to use IBM Network Authentication Service as a replacement technology for DCE, the following is required:

- IBM DCE Version 3.2 with PTF Set 3 or higher
- IBM Network Authentication Service Version 1.3 or higher
- An LDAP directory that is supported by both IBM DCE Version 3.2 or higher and IBM Network Authentication Service Version 1.3 or higher

© Copyright IBM Corp. 2003. All rights reserved.

4.1 Introduction

Authentication data is used to authenticate the identities of principals and is comprised of identity, key, and policy data. DCE and IBM Network Authentication Service use similar authentication data, and both support locating this data in an LDAP directory using common LDAP objects and attributes. This makes it possible to migrate DCE authentication data to an LDAP directory and then configure IBM Network Authentication Service to use the same data. After doing this, you can use the data with both DCE and IBM Network Authentication Service, or you can remove DCE-specific data from the LDAP directory and use the authentication data only with IBM Network Authentication Service.

Note: The configuration of IBM Network Authentication Service followed by DCE configuration is not supported. DCE must be configured to use LDAP prior to the configuration of IBM Network Authentication Service. Also, IBM Network Authentication Service cannot use DCE data that is not related to authentication. For example, IBM Network Authentication Service cannot use DCE data that is used to define groups, group membership, UUIDs, and extended registry attributes (ERAs) that are not used for authentication.

4.2 Migrating DCE data to an LDAP directory

To migrate DCE authentication data to the LDAP directory, refer to the *IBM DCE Version 3.2 Security Registry and LDAP Integration Guide* and follow the instructions for migrating the entire DCE registry database to the LDAP directory. Bear in mind that DCE does not support migrating only a portion of the DCE registry database to the LDAP directory.

4.3 Configuring IBM Network Authentication Service

To configure IBM Network Authentication Service to use the DCE authentication data that you migrated to the LDAP directory, do the following:

1. When you load the Kerberos LDIF schema file provided by IBM Network Authentication Service, be sure that all of the schema entries load without error. It might be necessary to do ldap modify operations in order to load some of the schema entries without error.
2. Configure IBM Network Authentication Service in the realm by following the instructions in the *IBM Network Authentication Service Version 1.3 for AIX,*

Administrator's and User's Guide. Use the LDAP plug-in configuration and observe the following exceptions:

- Do not configure a new realm entry. Instead, use the realm entry that you configured for DCE.
- In the realm entry, add one *krbTrustedAdmObject* attribute and one *krbKdcServiceObject* attribute for each IBM Network Authentication Service server in the realm, and configure the identity of the IBM Network Authentication Service server in each of these attributes.
- Update the access control configuration of the LDAP directory so that each IBM Network Authentication Service server, as well as each DCE server, has full permission to access attributes residing in the realm entry, under the realm entry, and under each entry referenced by the *krbPrincSubtree* attribute of the realm entry.
- Configure the appropriate LDAP bind protocol. For more information, refer to the *IBM Network Authentication Service Version 1.3 for AIX, Administrator's and User's Guide*.
- When running the **`config.krb5`** command or when editing the configuration files:
  - Specify a realm name that is the same as the DCE cell name stored in the *krbRealmName-v2* attribute of the realm entry.
  - You will not be prompted for a database master password (also referred to as a *master key*). The DCE master key will be the master key of the realm.

3. Configure clients in the realm by following the instructions in the *IBM Network Authentication Service Version 1.3 for AIX, Administrator's and User's Guide*.
4. Revise each IBM Network Authentication Service krb5.conf file, using a text editor, so that the enctype field lists des-cbc-crc as the first encryption type. This is necessary if sharing keys with DCE, because des-cbc-crc is the only encryption type supported by DCE. You must restart the KDC servers for the changes to take effect.

4.4 Managing the data in a shared environment

If you want DCE and IBM Network Authentication Service to use the same authentication data, you must observe the following special considerations for

managing this environment (and see sections 4.6, “Details about shared data” on page 79 and 4.7, “Details about non-shared data” on page 81):

- When creating a principal or policy object that contains shared data, use DCE interfaces to create the object. (An object created using IBM Network Authentication Service interfaces will not be recognized by DCE.)
- When deleting a principal or policy object that contains shared data, use DCE interfaces to delete the object. (IBM Network Authentication Service interfaces will not delete an object that contains DCE data.)
- When viewing a principal or policy object that contains shared data, use either the DCE or IBM Network Authentication Service interfaces to view the object. Regardless of which set of interfaces you use, you will be able to view all shared data. However, if the object contains any non-shared data, observe the following:
  - If you use DCE interfaces to view an object with non-shared data, you will be able to view only the data that is used by DCE and stored in the attributes used by DCE.
  - If you use IBM Network Authentication Service interfaces to view an object with non-shared data, you will be able to view only the non-shared data used by IBM Network Authentication Service and stored in the attributes used by IBM Network Authentication Service.
- When modifying a principal or policy object with shared data, use one of the following interfaces depending on the data in the object that you want to modify:
  - If you want to modify shared data, use either DCE or IBM Network Authentication Service interfaces to make the modifications. Understand that DCE or IBM Network Authentication Service can change the same shared data.
  - If you want to modify data that is not shared because it is DCE-specific, use DCE interfaces to make the modifications.
  - If you want to modify data that is not shared because it is IBM Network Authentication Service–specific, use IBM Network Authentication Service interfaces to make the modifications.
  - If you want to modify data that is not shared because DCE and IBM Network Authentication Service stores the data in different attributes, make the modification twice: one time with the DCE interfaces and the second time with the IBM Network Authentication Service interfaces (or in reverse sequence).
- The key version number is shared data. The administrator must ensure that the DCE and IBM Network Authentication Service keytab files are kept in sync with the key version number.

4.5 Removing DCE-specific data

If you no longer need to use DCE, you can remove DCE-specific data from the LDAP directory by doing the following:

1. Set the delete type in the realm entry to *dce* and un-configure DCE. Because the delete type has been set to *dce*, DCE will delete only DCE-specific data but not data used by IBM Network Authentication Service or any other applications.
2. In the realm entry, remove any of the following attributes that configure the identity of a DCE server:
   - krbTrustedAdmObject
   - krbKdcServiceObject
3. Modify the access control configuration of the LDAP directory so that the DCE servers can no longer access attributes residing in the realm entry and under the realm entry.
4. If you desire an encryption type other than des-cbc-crc, edit each IBM Network Authentication Service krb5.conf file so that the enctype field contains the desired encryption type.

After doing this, you must manage the data using IBM Network Authentication Service interfaces.

4.6 Details about shared data

Table 4-1 lists the authentication data that DCE and IBM Network Authentication Service can share because both use the data in the same way and store this data in the same LDAP attributes.

Table 4-1 Shared data

DCE Object	Network Authentication Object	Shared Data	Special Considerations
account	principal	account valid	
account	principal	allow/disallow forwardable tickets	
account	principal	allow/disallow postdated tickets	

DCE Object	Network Authentication Object	Shared Data	Special Considerations
account	principal	allow/disallow proxiable tickets	
account	principal	allow/disallow renewable tickets	
account	principal	allow/disallow server	
account	principal	allow/disallow subkey	
account	principal	allow/disallow TGT based authentication	
account	principal	associated DCE organization or IBM Network Authentication Service policy	
account	principal	current key version	
account	principal	expiration date	
account	principal	maximum renewable lifetime	
account	principal	maximum ticket lifetime	
key	key	key data	
key	key	key type	
key	key	key version	
key	key	version of master key used to encrypt this key	
key	key	salt/pepper type	
key	key	salt/pepper value	
master key	master key	master key file	
master key	master key	master key value	

DCE Object	Network Authentication Object	Shared Data	Special Considerations
master key	master key	master key version	
organization	policy	password lifetime	Possible conflict if a DCE ERA is used to set the password lifetime.
organization	policy	password minimum length	Possible conflict if a DCE ERA is used to set the password minimum length.
principal	principal	principal name	
realm	realm	name of master key file	
realm	realm	version of master key currently used in this realm	
realm	realm	name of master key principal	
realm	realm	name of realm	

4.7 Details about non-shared data

Some data cannot be shared by DCE and IBM Network Authentication Service for one of the following reasons:

- The data is specific to DCE.
- The data is specific to IBM Network Authentication Service.
- DCE and IBM Network Authentication Service store the data in different attributes.

Table 4-2 on page 82 lists the authentication data that DCE and IBM Network Authentication Service cannot share because the data is specific to DCE. In addition to the data in this table, IBM Network Authentication Service cannot use any DCE data that is not used for authentication, such as group, group membership, and UUID data.

Table 4-2 Data specific to DCE

DCE Object	Non-Shared Data	Notes to Administrator
account	creation_date	
account	creator	
account	key expiration time	
account	multiple key versions OK	
account	user-to-user authentication	
master key	key type	
realm	default certificate lifetime	
realm	password lifetime	
registry policy	passwd_min_len	

Table 4-3 lists the authentication data that cannot be shared because it is specific to IBM Network Authentication Service.

Table 4-3 Data specific to IBM Network Authentication Service

IBM Network Authentication Service Object	Non-Shared data	Notes to Administrator
policy	minimum password lifetime	
principal	e_data	
principal	last password change	
principal	new principal	
principal	password change service	
principal	principal type	When DCE creates a principal, it sets the principal type to an NT principal. DCE does not update or use this attribute.
principal	requires pre-authentication	
principal	requires hardware authentication	
principal	requires password change	

IBM Network Authentication Service Object	Non-Shared data	Notes to Administrator
principal	supports MD5	
principal	tl_data	

The authentication data in Table 4-4 cannot be shared because DCE and IBM Network Authentication Service store the data in different attributes.

Table 4-4 Data stored in different attributes

DCE Object	Network Authentication Object	Non-Shared Data	Notes to Administrator
account	principal	current number of bad login attempts	
account	principal	date of last modifier	
account	principal	name of last modifier	
account	principal	time of last bad login	
account	principal	time of last good login	
organization ERA	realm	password dictionary files	
organization ERA (IBM-pwd_comp_rules: mindiff)	policy	minimum different characters in password	When this data is created or modified using DCE interfaces, the data is copied to the attribute used by IBM Network Authentication Service. A possible conflict could occur if this same DCE ERA is configured for a principal.

DCE Object	Network Authentication Object	Non-Shared Data	Notes to Administrator
organization ERA (IBM-pwd_hist_rules: histsize)	policy	number of password history entries	When this data is created or modified using DCE interfaces, the data is copied to the attribute used by IBM Network Authentication Service. A possible conflict could occur if this same DCE ERA is configured for a principal.
realm	realm	maximum renewable lifetime	
realm	realm	maximum ticket lifetime	

5

Using DCE objects with IBM Tivoli Access Manager

The information in this chapter is for administrators who have IBM DCE Version 3.2 and IBM Tivoli Access Manager (formerly known as Tivoli Policy Director) and want to do either of the following:

- Use DCE and IBM Tivoli Access Manager to share user and group objects in LDAP.
- Migrate DCE user and group information into existing IBM Tivoli Access Manager objects in LDAP.
- Migrate DCE user and group information in the IBM Directory to IBM Tivoli Access Manager objects.

© Copyright IBM Corp. 2003. All rights reserved.

5.1 Introduction

IBM DCE 3.2 and IBM Tivoli Access Manager both support using an LDAP directory to store data. This makes it possible for DCE and IBM Tivoli Access Manager to share objects in LDAP. That is, DCE users and groups can have the same Distinguished Name (DN) as IBM Tivoli Access Manager users and groups. Otherwise, little is shared.

Specifically, the following are not shared:

- Passwords
- Unique IDs
- User attributes
- Most group attributes

The group membership attribute, however, can be shared:

- Group membership attribute

Policy data can be shared in a limited way because policies are handled differently between the technologies. DCE allows policies to be created, then users are associated with a specific policy. IBM Tivoli Access Manager creates a default policy, then users can override that policy. The user overrides are stored with the user's definition. A policy can be shared between IBM Tivoli Access Manager and DCE if DCE attaches to IBM Tivoli Access Manager's default policy. This is discussed in 5.4, “Managing objects in a shared environment” on page 96.

The following policy data can be shared if DCE is attached to the IBM Tivoli Access Manager default policy:

- Account lifetime
- Password expiration time
- No spaces in a password
- No alpha characters in a password
- Maximum lifetime of a password
- Minimum lifetime of a password
- Account expiration lockout time
- Maximum number of failed logins

5.2 Data representation

The following figures illustrate how DCE and IBM Tivoli Access Manager store and share data in the LDAP directory.

Figure 5-1 shows a sample DCE principal used by DCE as it is stored in the LDAP directory tree.

Figure 5-1 Sample DCE principal called user1 in LDAP

Figure 5-2 shows a sample user and its representation in the LDAP directory tree as it is stored by the IBM Tivoli Access Manager.

Figure 5-2 Sample IBM Tivoli Access Manager user called user1 in LDAP

Figure 5-3 illustrates how a DCE account and an IBM Tivoli Access Manager user share an object in the IBM Tivoli Access Manager tree.

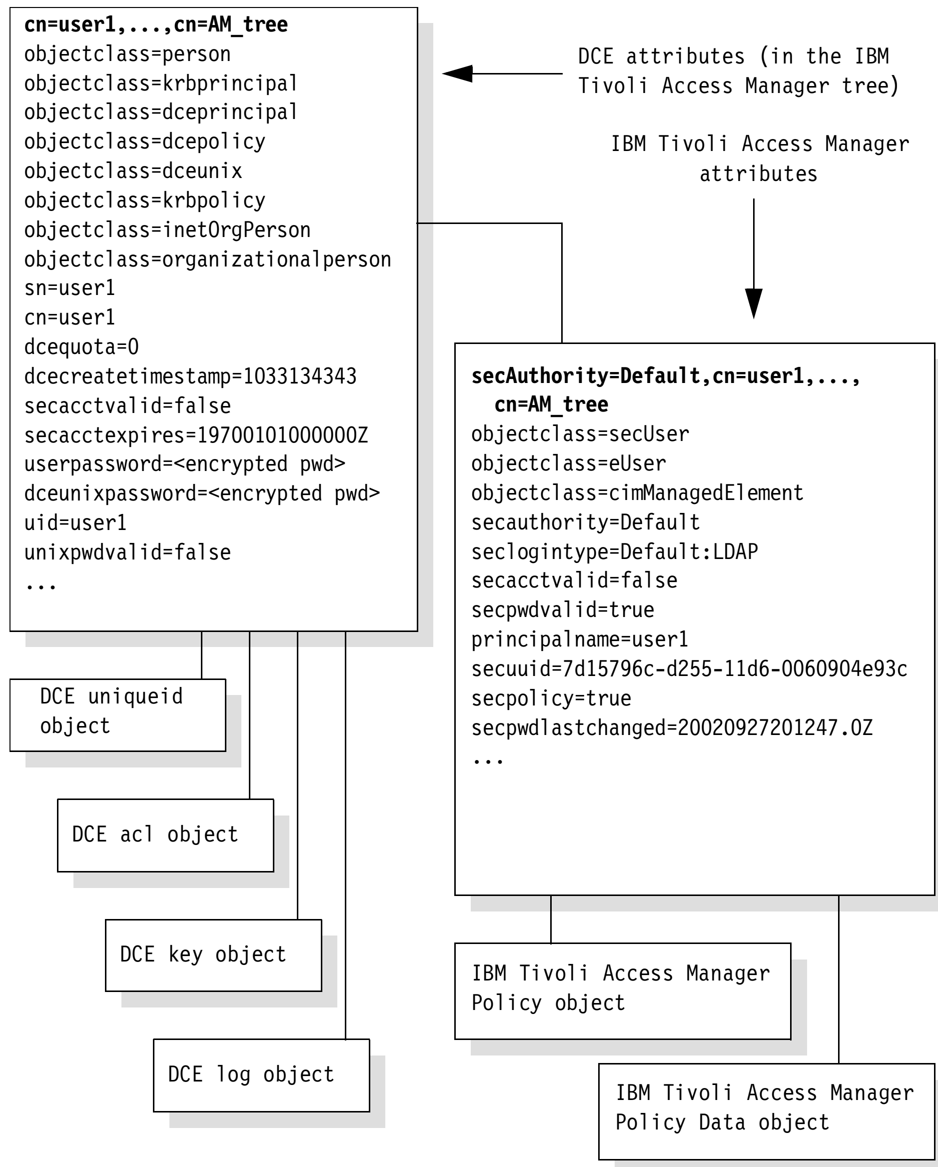


Figure 5-3 Sample shared account called user1

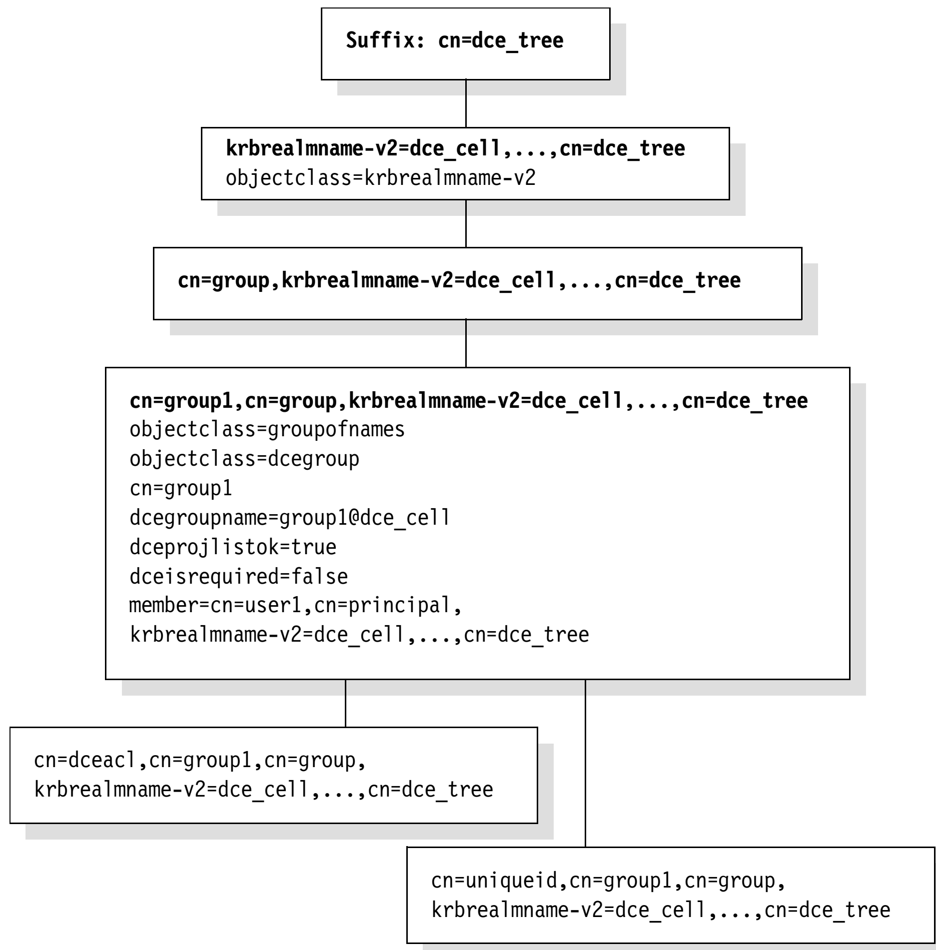


Figure 5-4 Sample DCE group called group1 in LDAP

Figure 5-4 shows a sample representation of a DCE group within the DCE tree of the LDAP directory, and Figure 5-5 on page 91 shows how the IBM Tivoli Access Manager stores a group in its LDAP directory tree.

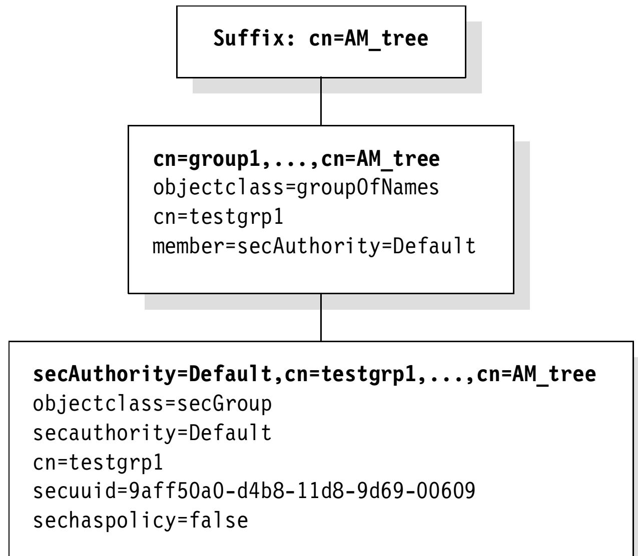


Figure 5-5 Sample IBM Tivoli Access Manager group called group1 in LDAP

Figure 5-6 represents the IBM Tivoli Access Manager directory tree of a sample group that is shared between DCE and the IBM Tivoli Access Manager.

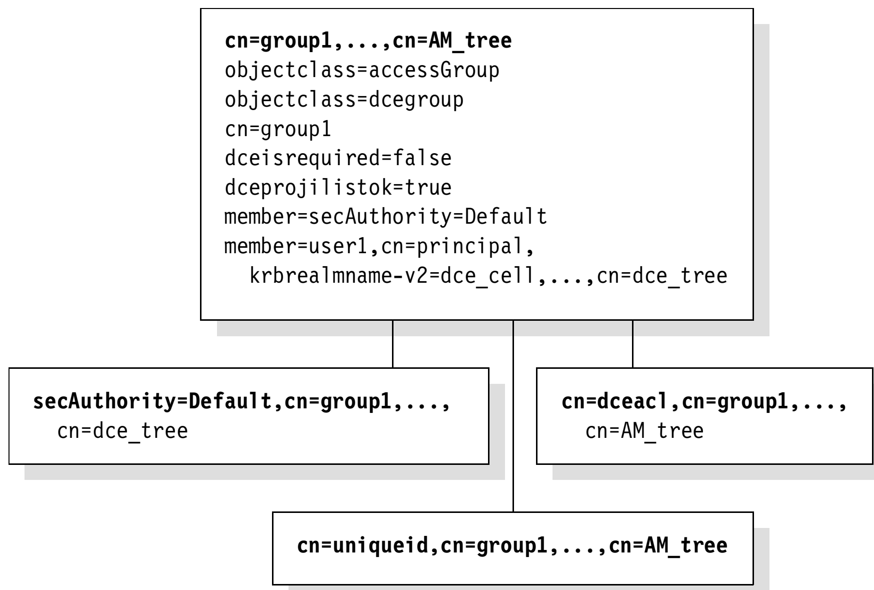


Figure 5-6 Sample shared group called group1

5.3 Configuration scenarios

In the sections that follow, three practical scenarios are discussed as they could exist in practical environments.

5.3.1 Scenario 1

The DCE user data has been migrated to LDAP already. The administrator wants to configure IBM Tivoli Access Manager and attach to the existing DCE principals and groups.

Currently, it is not possible to configure IBM Tivoli Access Manager in an LDAP server that DCE uses. When IBM Tivoli Access Manager configures, it attempts to change or update some attributes. However, some of these attributes are used by DCE. This causes the IBM Tivoli Access Manager configuration to fail because attributes cannot be changed when they are in use.

5.3.2 Scenario 2

A DCE registry database is being migrated into an existing LDAP tree to share objects with IBM Tivoli Access Manager. That is, DCE principals and groups are migrated into existing IBM Tivoli Access Manager user and group objects in LDAP. To migrate DCE:

1. Load the DCE and Kerberos schema files into the LDAP directory as described in the *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.
2. Using the IBM Tivoli Access Manager interfaces, create an IBM Tivoli Access Manager user for each DCE user that needs to share an object in LDAP. For each of these users, add the krbprincipal objectclass and krbprincipalname attribute to the user object in LDAP.

   To add the IBM Tivoli Access Manager user, type the following:

   ```
   pdadmin> user create <user name> <user DN> <CN> <SN> <user password>
   ```

   To add krbprincipal and krbprincipalname attributes, complete the following substeps:

   a. Type the following:

   ```
   ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <user DN> \
   objectclass=person objectclass > personobj
   ```

   b. Edit the personobj file created and append the following:

   ```
   objectclass=krbprincipal
   krbprincipalname=<princ>@<realm>
   ```

c. Type the following:

```
ldapmodify [-h <host>] -D <bind DN> -w <bind passwd> -f personobj
```

d. Verify that the krbprincipal objectclass and the krbprincipalname attribute were added by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <user DN> \
objectclass=person
```

3. Using the IBM Tivoli Access Manager interfaces, create an IBM Tivoli Access Manager group for each DCE group that needs to share an object in LDAP. For each of these groups, add the dcegroup objectclass and the dcegroupname attribute to the group object in LDAP.

   To add the IBM Tivoli Access Manager group, type the following:

```
pdadmin> group create <group name> <DN> <CN>
```

   To add dcegroup and dcegroupname, complete the following substeps:

   a. Type the following:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <group DN> \
objectclass=groupOfNames > groupobj
```

   b. Edit the groupobj file created and append the following:

```
objectclass=dcegroup
dcegroupname=<group>@<realm>
```

   c. Type the following:

```
ldapmodify [-h <host>] -D <bind DN> -w <bind passwd> -f groupobj
```

   d. Verify that the dcegroup objectclass and dcegroupname attribute were added by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <group DN> \
objectclass=dcegroup
```

4. Add the DCE realm information to LDAP as described in the *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*. The subtree or subtrees containing the IBM Tivoli Access Manager users and groups need to be added the krbprincsubtree attribute in the DCE realm information. To add the subtree or subtrees:

   a. Type the following:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <realm DN> \
objectclass=krbrealm-v2 krbprincsubtree > subtree
```

   b. Edit the subtree file that is created and append a line for each additional subtree needed:

   krbprincsubtree=<subtree DN1>
   krbprincsubtree=<subtree DN2>

c. Type the following:

```
ldapmodify [-h <host>] -D <bind DN> -w <bind passwd> -f subtree
```

d. Verify that the krbprincsubtree attributes are added by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <realm DN> \
objectclass=krbrealm-v2
```

5. Using the DCE interfaces, migrate the legacy DCE registry database to LDAP as described in the *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.
6. When migration is finished, verify that the user and group objects are shared in LDAP by typing either of the following:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <user DN> \
objectclass=krbprincipal
```

or

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <user DN> \
objectclass=dcegroup
```

5.3.3 Scenario 3

Legacy DCE is migrated to LDAP and IBM Tivoli Access Manager is already configured in LDAP. However, you still must attach IBM Tivoli Access Manager users to DCE users. Scenario 3 differs from scenario 2 in that DCE is not migrated into an existing IBM Tivoli Access Manager configuration.

To attach the IBM Tivoli Access Manager users to DCE users:

1. Using the DCE interfaces, migrate the legacy DCE registry database to LDAP as described in the *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.
2. Verify that cn=SecurityGroup,secAuthority=Default, which is an LDAP group access control list (ACL), has been added to the suffix where DCE data resides. This is done when IBM Tivoli Access Manager is configured in order to enable IBM Tivoli Access Manager to write in the DCE subtrees.
3. For each DCE principal that needs to share an LDAP object with IBM Tivoli Access Manager, use the IBM Tivoli Access Manager interface to create the user. To create the user, use the DN of the DCE principal:

   a. First, add the inetOrgPerson objectclass to the principal object in LDAP by completing the following substeps:

      i. Run the following command:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <realm DN> \
"(&(objectclass=krbprincipal)(krbprincipalname=<princ>@<realm>))" \
objectclass > princobj
```

ii. Edit the princobj file and append the following:

```
objectclass=inetOrgPerson
```

iii. Add the objectclass to the principal object by typing:

```
ldapmodify [-h <host>] -D <bind DN> -w <bind passwd> -f princobj
```

iv. Verify that the objectclass was added by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <realm DN> \
"(&(objectclass=krbprincipal)(krbprincipalname=<princ>@<realm>))" \
objectclass
```

> **Note:** IBM Tivoli Access Manager Version 3.9 requires this addition of the inetOrgPerson objectclass; otherwise the **`pdadmin user import`** command (see below) will fail. Newer versions or releases of the IBM Tivoli Access Manager may not require this modification.

b. Get the DN of the user object by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <realm DN> \
“(&(objectclass=krbPrincipal)(krbPrincipalName=<princ>@<realm>))”
```

c. Add the IBM Tivoli Access Manager user by typing:

```
pdadmin> user import <user name> <user DN>
```

d. Verify that the IBM Tivoli Access Manager user was created by typing:

```
pdadmin> user show <user name>
```

e. Verify that the user information is in LDAP along with the DCE principal information by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <user DN> \
objectclass=krbprincipal
```

4. For each DCE group that needs to share an LDAP object with IBM Tivoli Access Manager, use the IBM Tivoli Access Manager interfaces to create the group using the DN of the DCE group:

a. Get the DN of the group object by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <realm DN> \
"(&(objectclass=dcegroup)(dcegroupname=<group name>@realm))"
```

b. Attach an IBM Tivoli Access Manager group to the DCE group by typing:

```
pdadmin> group import <group name> <group DN>
```

c. Verify that the IBM Tivoli Access Manager group is created:

```
pdadmin> group show <group name>
```

d. Verify that the IBM Tivoli Access Manager information and DCE information are in LDAP:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -D <group DN> \
objectclass=*
```

5.4 Managing objects in a shared environment

This section describes how to manage shared objects in LDAP when using both DCE and IBM Tivoli Access Manager.

5.4.1 Creating a user with IBM Tivoli Access Manager

You can create an object with IBM Tivoli Access Manager and attach a DCE principal by completing the following steps:

1. Create a user with IBM Tivoli Access Manager by typing:

```
pdadmin> user create <user name> <user DN> <CN> <SN> <user password>
```

2. Add the krbprincipal objectclass and the krbprincipalname attribute to the user's DN by completing the following substeps:

   a. Type the following:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <user DN> \
objectclass=person objectclass > userobj
```

   b. Edit the userobj file that was created in the previous step and add the following lines to the end of the file:

   objectclass=krbprincipal
   krbpirncipalname=<user>@<realm>

   c. Add the objectclasses and attributes to the user object by typing:

```
ldapmodify [-h <host>] -D <bind DN> -w <bind passwd> -f userobj
```

   d. Enter another LDAP search to verify that the object is updated. This can be verified by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <user DN> \
objectclass=krbprincipal
```

3. Create the DCE principal by typing:

```
dcecp> princ create <user name>
```

4. Verify that the user has been created by typing:

```
dcecp> princ show <user name>
```

5. Verify that DCE information is in LDAP by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <user DN> \
objectclass=*
```

5.4.2 Creating a group with IBM Tivoli Access Manager

You can create a group with IBM Tivoli Access Manager and attach a DCE group by completing the following steps:

1. Create a group with IBM Tivoli Access Manager by typing:

```
pdadmin> group create <group name> <DN> <CN>
```

2. Add the dcegroup objectclass and the dcegroupname attribute to the group DN. This can be accomplished by completing the following substeps:

   a. Type the following:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <group DN> \
objectclass=accessgroup objectclass > groupobj
```

   b. Edit the groupobj file created and add these lines to the end of the file:

```
objectclass=dcegroup
dcegroupname=<group>@<realm>
```

   c. Add the objectclass and attribute to the user object:

```
ldapmodify [-h <host>] -D <bind DN> -w <bind passwd> -f groupobj
```

   d. Enter another ldap search to make sure the object is updated by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <user DN> \
objectclass=dcegroup
```

3. Create the DCE group by typing:

```
dcecp> group create <group name>
```

4. Verify that the group is created by typing:

```
dcecp> group show <group name>
```

5. Verify that DCE information is in LDAP by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passsword> -b <group DN> \
objectclass=*
```

5.4.3 Adding a member to a group using IBM Tivoli Access Manager

The group membership attribute is shared between IBM Tivoli Access Manager and DCE. When a user is added to the group using IBM Tivoli Access Manager, the group membership list is also seen by DCE. DCE will report this user as a member of the group as long as the user is a valid DCE user. This causes some inconsistency in DCE because users keep a list of their groups. When a user is

added to a group using IBM Tivoli Access Manager, the DCE user is not aware of the fact that they have been added to a group. Thus, it is recommended that users are added to groups using DCE. This method is described in 5.4.9, “Adding a member to a group using DCE” on page 100. However, to add a member to a group with IBM Tivoli Access Manager, type the following:

```
pdadmin> group modify <group name> add <user name>
```

5.4.4 Deleting a user using IBM Tivoli Access Manager

To delete a user from IBM Tivoli Access Manager, type:

```
pdadmin> user delete <user name>
```

This will remove the IBM Tivoli Access Manager user definition in LDAP. The person object that the IBM Tivoli Access Manager user is attached to is not changed. Therefore, the user remains defined in DCE. The IBM Tivoli Access Manager **delete** command has another flag, **-registry**, which attempts to delete the person object:

```
pdadmin> user delete -registry <user name>
```

When this command is issued, IBM Tivoli Access Manager attempts to delete the person object. If subtrees that IBM Tivoli Access Manager does not know about are attached to the person object, then it will not remove that person object. DCE creates subtrees under the person object; this prevents IBM Tivoli Access Manager from removing the person object. The DCE person is not changed.

5.4.5 Deleting a group using IBM Tivoli Access Manager

To delete a group using IBM Tivoli Access Manager, type:

```
pdadmin> group delete <group name>
```

This command removes the IBM Tivoli Access Manager group definition from LDAP. All of the IBM Tivoli Access Manager attributes will be removed from the group object. The DCE group attributes and objectclass are not changed. The **-registry** flag on the **delete** command will attempt to delete the group object. To delete the group object, type:

```
pdadmin> group delete -registry <group name>
```

If subtrees that IBM Tivoli Access Manager does not know about are attached to the group object, then it will not remove that group object. DCE creates subtrees under the group object; this prevents IBM Tivoli Access Manager from removing the group object. The DCE group is not changed.

5.4.6 Removing a member from an IBM Tivoli Access Manager group

The group membership attribute is shared between IBM Tivoli Access Manager and DCE, so when a user is deleted from a group using IBM Tivoli Access Manager, the user is no longer listed as a member of the DCE group. That is, when you issue the command to see the DCE list group membership, the user does not appear. The problem is that the DCE user keeps a list of the groups it belongs to. It appears to the DCE user that it is still a member of the group. Because of this, it is recommended that group membership be done using the DCE utilities as described in 5.4.12, "Removing a member from a group with DCE commands" on page 102.

5.4.7 Creating a principal with DCE

To create a principal with DCE and then attach it to IBM Tivoli Access Manager, complete the following steps:

1. Create a principal with DCE by typing:

   ```
   dcecp> princ create <principal name>
   ```

2. Add the inetOrgPerson objectclass to the principal object in LDAP by completing the following substeps:

   a. Run the following command:

      ```
      ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <realm DN> \
      "(&(objectclass=krbprincipal)(krbprincipalname=<princ>@<realm>))" \
      objectclass > princobj
      ```

   b. Edit the princobj file and append the following:

      ```
      objectclass=inetOrgPerson
      ```

   c. Add the objectclass to the principal object by typing:

      ```
      ldapmodify [-h <host>] -D <bind DN> -w <bind passwd> -f princobj
      ```

   d. Verify that the objectclass was added by typing:

      ```
      ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <realm DN> \
      "(&(objectclass=krbprincipal)(krbprincipalname=<princ>@<realm>))" \
      objectclass
      ```

> **Note:** IBM Tivoli Access Manager Version 3.9 requires this addition of the inetOrgPerson objectclass; otherwise the **pdadmin user import** command (see below) will fail. Newer versions or releases of the IBM Tivoli Access Manager may not require this modification.

3. Get the DN of the user object by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <realm DN> \
“(&(objectclass=krbPrincipal)(krbPrincipalName=<princ>@<realm>))”
```

4. Add the IBM Tivoli Access Manager user by typing:

```
pdadmin> user import <user name> <user DN>
```

5. Verify that the IBM Tivoli Access Manager user was created by typing:

```
pdadmin> user show <user name>
```

6. Verify that the user information is in LDAP along with the DCE principal information by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <user DN> \
objectclass=krbprincipal
```

5.4.8 Creating a DCE group

To create a DCE group and then attach it to a IBM Tivoli Access Manager group, complete the following steps:

1. Create a group with DCE by typing:

```
dcecp> group create <group name>
```

2. Get the DN of the group object by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -b <realm DN> \
"(&(objectclass=dcegroup)(dcegroupname=<group name>@realm))"
```

3. Attach an IBM Tivoli Access Manager group to the DCE group by typing:

```
pdadmin> group import <group name> <group DN>
```

4. Verify that the IBM Tivoli Access Manager group is created by typing:

```
pdadmin> group show <group name>
```

5. Verify that the IBM Tivoli Access Manager information and DCE information are in LDAP by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind passwd> -D <group DN> \
objectclass=*
```

5.4.9 Adding a member to a group using DCE

The group membership attribute is shared between DCE and IBM Tivoli Access Manager. When a member is added to a group with DCE, that membership list is also seen by IBM Tivoli Access Manager. If that user is a valid IBM Tivoli Access Manager user, then IBM Tivoli Access Manager considers the user a member of that group.

The following command adds a member to a group in DCE:

```
dcecp> group add <group name> -member <user name>
```

5.4.10 Deleting a user using DCE

To delete a principal using DCE, type the following:

```
dcecp> princ delete <user name>
```

This will delete a principal. The information that gets removed from LDAP depends on the krbdeletetype attribute value. The krbdeletetype values are:

none(1) This value indicates that nothing will be deleted from LDAP. The delete command is successful, but nothing is deleted.

dce(2) This value indicates that all of the DCE-related attributes are removed from the user object in LDAP. The IBM Tivoli Access Manager user is still defined and usable.

dce and krb5(3) This value indicates that all of the DCE- and Kerberos-related attributes are removed from the person object in LDAP. Also, all of the DCE- and Kerberos-related subtrees attached to the person are removed. The IBM Tivoli Access Manager user is still defined and usable.

all(4) This value indicates that the person object and all of the DCE and Kerberos subtrees are removed. An error is returned because the person object can not be removed. There are IBM Tivoli Access Manager related subtrees attached to the person object and that prevents DCE from deleting the object. From an IBM Tivoli Access Manager perspective, the user is defined and usable.

5.4.11 Deleting a group using DCE commands

To delete a group using DCE, type the following:

```
dcecp> group delete <group name>
```

This command deletes a group. The information that is removed from LDAP is based on the krbdeletetype attribute value. The krbdeletetype values are:

none(1) This value indicates that nothing is deleted from LDAP. The delete command is executed successfully, but the group is not deleted from LDAP. The IBM Tivoli Access Manager group is defined and usable.

dce(2) — This value indicates that all of the DCE-related attributes are removed from the group object in LDAP. The IBM Tivoli Access Manager group is still defined and usable.

dce and krb5(3) — This value indicates that all of the DCE-related attributes are removed from the group object in LDAP. The IBM Tivoli Access Manager group is still defined and usable.

all(4) — This value indicates that the group object and all of the DCE-related subtrees are removed from LDAP. When DCE attempts to delete the group object, the IBM Tivoli Access Manager related subtrees that are on the group object make its removal impossible. From an IBM Tivoli Access Manager perspective, the group is defined and usable.

5.4.12 Removing a member from a group with DCE commands

The group membership attribute is shared between IBM Tivoli Access Manager and DCE. When a user is deleted from a group using DCE commands, that user is no longer listed as a member of the IBM Tivoli Access Manager group.

5.4.13 Sharing policies

Only one policy can be shared between DCE and IBM Tivoli Access Manager. IBM Tivoli Access Manager has a default policy; all variations to the default policy are placed on the user object. The only way for DCE and IBM Tivoli Access Manager to share a policy is for DCE to attach to the IBM Tivoli Access Manager default policy. The IBM Tivoli Access Manager default policy's DN is cn=Default,cn=Policies,secAuthority=Default.

5.4.14 Attaching a DCE policy

To attach a DCE policy to the IBM Tivoli Access Manager policy, complete the following steps:

1. Put the IBM Tivoli Access Manager Policy in the principal subtree list on the DCE realm by completing the following substeps:
   a. Type the following:
   ```
   ldapsearch [-h <host>] -D <bind DN> -w <bind pwd> -b <DCE realm DN> \
   objectclass=krbrealm-v2 krbprincsubtree > subtreefile
   ```
   b. Edit the subtreefile file and add the following line to the end of it:

      krbprincsubtree=cn=Default,cn=Policies,secAuthority=Default

c. Add the new principal subtree by typing:

```
ldapmodify [-h <host>] -B <bind DN> -w <bind pwd> -f <subtreefile>
```

d. Verify that the principal subtree was added by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind pwd> -b <DCE realm DN> \
objectclass=krbrealm-v2 krbprincsubtree
```

2. Add the krbpolicy objectclass and krbprolicyname attribute to the IBM Tivoli Access Manager's default policy by completing the following steps:

a. Type the following:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind pwd> -b \
cn=Default,cn=Policies,secAuthority=Default objectclass=secPolicy \
objectclass > dfltpol
```

b. Edit the dfltpol file and add the following lines to the end of the file:

```
objectclass=krbpolicy
krbpolicyname=<policy name>@<realm>
```

c. Add the values to the default policy by typing:

```
ldapmodify [-h <host>] -D <bind DN> -w <bind pwd> -f dfltpol
```

d. Verify that the objectclass and attribute were added successfully by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind pwd> -b \
cn=Default,cn=Policies,secAuthority=Default objectclass=secPolicy
```

3. Create the DCE policy by typing:

```
dcecp> org create <policy name>
```

4. Verify that the DCE policy is attached to the IBM Tivoli Access Manager default policy by typing:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind pwd> -b \
cn=Default,cn=Policies,secAuthority=Default objectclass=secPolicy \
objectclass=*
```

If dcepolicy and dceorg objectclasses are now a part of the object, then the DCE *org* was created successfully.

5.4.15 Deleting a shared DCE policy

Deleting the DCE policy that is shared with the IBM Tivoli Access Manager default policy can result in the deletion of the IBM Tivoli Access Manager default policy.

Set the DCE delete type to **none** if DCE is attached to the IBM Tivoli Access Manager default policy. The following is a list of krbdeletetype values:

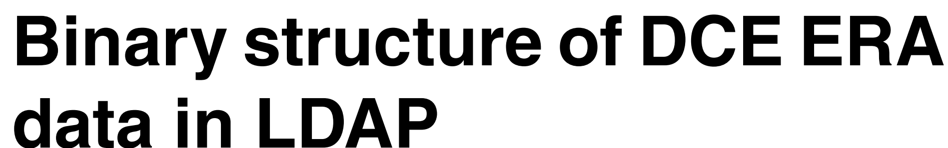
none(1)
: This value indicates that nothing is deleted from LDAP. The delete command is executed successfully but the policy is not removed from LDAP.

dce(2)
: This value indicates that all of the DCE-related attributes are removed from the policy. The problem is that some of the DCE-related attributes are shared with IBM Tivoli Access Manager. Thus, some attributes will be missing from the IBM Tivoli Access Manager Policy. All of the DCE-related subtrees are removed from the policy object.

dce and krb(3)
: This value indicates that all of the DCE- and Kerberos-related attributes are removed from the policy object. Most of these attributes are shared with IBM Tivoli Access Manager. This leaves the IBM Tivoli Access Manager policy object without policy attributes.

all(4)
: This value indicates that the entire policy object is removed. The IBM Tivoli Access Manager is without a default policy, which causes problems when IBM Tivoli Access Manager creates users.

6



Binary structure of DCE ERA data in LDAP

This chapter describes how to read binary DCE ERA data that has been migrated to an LDAP directory, such as IBM Directory Server, using non-DCE interfaces. This information is provided as a reference for environments that use DCE ERA data (for example, for storing additional policy data), have migrated the DCE registry database to an LDAP directory, and have removed DCE from their configuration. These environments might need to know how to read the migrated DCE ERA data using non-DCE interfaces. If your environment does not use DCE ERA data or if you do not need to read migrated DCE ERA data using non-DCE interfaces, you may skip this chapter.

The following is covered in this chapter:

- A brief recap of the DCE-to-LDAP migration process
- A description of how to read the binary DCE ERA data migrated to LDAP

© Copyright IBM Corp. 2003. All rights reserved.

6.1 Recap: The DCE to LDAP migration process

As a short recap, a recommended replacement for the DCE registry database is the IBM Directory Server that is the IBM implementation of a standardized LDAP directory. During the migration, a copy of the DCE registry database is stored and maintained in the LDAP directory where it is available to other services, such as the IBM Network Authentication Service or the IBM Tivoli Access Manager, for providing authentication and authorization services.

The document *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide* explains how DCE-related objects from the registry database are stored in the LDAP directory. User-defined Extended Registry Attributes (ERA) are stored in the LDAP directory as binary objects in a separate ERA structural object. The internal structure of this binary object is explained in the next section.

6.2 Reading binary DCE ERA data in LDAP

To read binary DCE ERA data migrated to an LDAP directory:

1. Obtain the sec_attr_base.h file that is shipped with IBM DCE Version 3.2, and include this file in your application.
2. Search the LDAP directory for the *PGO entry*, which is the entry representing the DCE principal, group, or organization for which the ERA is configured. To search for the PGO entry, set the LDAP search filter to one of the following, depending on whether the search is for a principal, group, or organization:

   ```
   krbPrincipalName = <principal name>@<realm_name>
   dceGroupName = <group name>@<realm_name>
   krbPolicyName = <organization name>@<realm_name>
   ```

   For example, to search for an entry representing a DCE principal with a principal name of john and a realm (cell) name of realm1, set the LDAP search filter to search for an LDAP attribute named krbPrincipalName with a value equal to john@realm1.
3. Search the LDAP directory for the *ERA entry*, which represents the ERA that has been configured for the DCE principal, group, or organization. To search for this ERA entry, set the LDAP search filter to search for an entry residing under the PGO entry where:

   ```
   object class = DCEERA
   ```

   and

   ```
   dceXattrName = <ERA name>
   ```

(This search filter could be combined with the search filter in the previous step.)

4. From the ERA entry, get the value stored in dceXattrValue attribute. The dceXattrValue is a binary LDAP attribute. DCE uses this attribute to store an era_object_data_t buffer, which is defined as follows:

```
typedef struct era_object_data_t {
   unsigned32        num_vals;
   unsigned32        data_len;
   uuid_t            era_uuid;
   unsigned char     db_attrs[1];
}  era_object_data_t;
```

   where:

num_vals	Number of rsdb_attr_object_t buffers stored in the db_attrs field of this era_object_data_t buffer.
data_len	Length of this era_object_data_t buffer.
era_uuid	UUID of the ERA defined in this era_object_data_t buffer.
db_attrs	One or more rsdb_attr_object_t buffers. Each rsdb_attr_object_t buffer contains an ERA value.

5. Use each rsdb_attr_object_t buffer to get each ERA value that is stored in this ERA object. The rsdb_attr_object_t buffer is defined as follows:

```
typedef struct rsdb_attr_inst_t {
    rsdb_thread_hdr_t   threads;
    unsigned32          attr_inst_len;
    sec_rgy_domain_t    domain;
    rsdb_pvt_id_t       object_id;
    sec_attr_encoding_t encoding;
    char                buf[VARYING];
} rsdb_attr_inst_t;
```

   where:

threads	Not used.
attr_inst_len	Length of this rsdb_attr_object_t buffer.
domain	Not used.
object_id	Not used.
encoding	The encoding type of the ERA value. The encoding type is defined in the sec_attr_encoding_t variable of the DCE sec_attr_base.h file.
buf	Buffer containing the ERA value. The data type of this value is defined by the encoding type.

Part 2

Replacement sample scenarios

In this part we provide example scenarios that are being migrated from DCE to other technologies.

© Copyright IBM Corp. 2003. All rights reserved.

7

Common replacement considerations

This chapter contains information that is common for all of the migration scenarios described in the chapters that follow. In particular, this chapter lists and explains:

- How to read the sample scenarios
- Common assumptions in the sample scenarios
- Simplifications in the sample scenarios
- Migration security considerations
- Performance considerations
- Considerations when using an LDAP directory
- SSL implementation hints

© Copyright IBM Corp. 2003. All rights reserved.

7.1 How to read the example scenarios

The example scenarios that are described in the following chapters of this section are all structured in the same manner:

- First, an introduction is given that explains the environment of the scenario and the strategy that is chosen for removing the DCE dependencies.
- A sample application program that depends on DCE services is explained, followed by an explanation of the way the DCE dependencies are removed in the revised application.
- The replacement roadmap is explained, including some brief steps to install and configure replacement products.
- As a result of the replacement, the application program using the replacement technologies is then explained.
- The scenario is summarized with a discussion that also elaborates on additional considerations and possible alternatives.

For the description of the code examples, only excerpts or fragments of the code are being shown and explained. The full program code listings are in the appendix of this book and can be downloaded. (See Appendix E, “Additional material” on page 417 for instructions.)

7.2 Common assumptions in the sample scenarios

The example applications used in the following chapters of this book are used for the purpose of explaining the use of the chosen replacement technologies. They do not represent real applications, as the application logic is considered irrelevant for the purpose of this book.

Because actual system and development environments vary to a large extent, only information that is assumed important in regard to the environment used for writing this book is provided.

7.3 Simplifications in the sample scenarios

The purpose of the replacement scenarios in this book is to demonstrate the technologies for the replacement of DCE dependencies. The examples are kept simple, and they are not meant to serve as schoolbook examples for production-ready applications.

Simplifications made in the scenarios include:

- The DCE-dependent application servers do not perform a login to DCE. Instead, they inherit their DCE credentials from the environment.
- There is no DCE ticket-refreshing logic included in the scenarios.
- Unless used by the underlying services, the example scenarios use a single thread approach.
- Although strongly recommended, SSL is not used to protect inter-system communication. (Also see section 7.7.3, “Using SSL in the replacement scenarios” on page 117.)
- The examples do not include any meaningful error-handling or error-recovery procedures, but rather terminate in the case of an error.

7.4 Security considerations

DCE provides both an environment for cross-platform distributed applications and a high level of security with respect to the following two areas:

- All DCE-internal communication and services can be highly secured, representing in its entirety a highly secure middleware platform.
- DCE applications are provided with a rich set of security-related functions, such as authentication and authorization services, and encryption for over-the-wire communication.

While it is the responsibility of the application architect and programmer to replace the security-related functions by means of other technology, the system administrator must ensure that the replacing infrastructure also maintains an acceptable level of security.

It is important that the security features of the replacement technologies (and the products that implement those technologies) are fully understood. This book is not intended as a replacement for proper training and education on the products being used.

For security considerations specific to SSL, please also read section 7.7, “SSL implementation hints” on page 116.

7.5 Performance considerations

This book and the examples provided with it do not take any performance issues into consideration. The primary purpose of the examples is to show replacement technologies and strategies.

If performance is of concern, the individual replacement technologies and the products that implement those technologies must be evaluated further for performance-related matters.

For example, the IBM Directory Server that is used as a replacement strategy for DCE security registry data provides several features for performance, such as:

- Excellent vertical scalability by supporting a wide range of industry platforms, including all IBM operating system platforms from Windows to z/OS™.
- Excellent horizontal scalability, not only by relying on the high performance IBM DB2® database, but also by providing replication for spreading a directory among several servers.

7.6 Using an LDAP directory

The replacement scenarios in the following chapters often use an LDAP directory for storing various kinds of information, such as the DCE security registry. In contrast to proprietary directories as maintained by DCE, an LDAP directory is an open-standard, multi-purpose directory that is likely to be installed and running on a dedicated system (or multiple systems if replication is being used) on the network. Because of this and some other differences to the DCE directories, it is worth taking a closer look at some features that an LDAP directory provides.

7.6.1 LDAP security considerations

Except for some administrative tasks, an LDAP directory is accessed over the network through a well-standardized protocol. There are many LDAP clients available, including standard Web browsers such as the Netscape browser products and Microsoft Internet Explorer. In other words, an LDAP directory is accessible over the network by potentially everybody who can establish a TCP/IP connection to the system that runs the LDAP directory. In most organizations, this includes most employees. Due to this, there is a security exposure if the information in the LDAP directory is not properly secured.

The IBM Directory Server offers several options to protect the directory contents from unauthorized access, including:

- Various authentication models, from anonymous to certificate-based mutual authentication over SSL
- Access Control Lists (ACL) for the protection of specific information in the Directory Information Tree (DIT)
- Audit trails

In addition to the protection options listed above, the system environment must be protected like every other system that stores sensitive information, including:

- Strict account and password management for administrator accounts
- Securing access to the underlying IBM DB2 database
- Physical and logical access control, which includes adequate protection of any backup media
- Network segregation: placing sensitive systems in network segments that are protected with firewalls or other similar network access control

7.6.2 Availability and performance considerations

Another important feature of the IBM Directory Server is replication. Note that replication is not part of the LDAPv3 standard; therefore, vendor products have different implementations. The IBM Directory Server supports single-master and multiple-master replication. In the more-frequently used single master replication mode, one IBM Directory Server acts as a master, being the only directory that allows updates to the stored information. One or more LDAP replica servers receive their updates only from the master. Except for this master/replica update, the LDAP replica servers do not allow any updates, but rather refer a client to the master if the client wishes to update the directory.

Just as in DCE, replication serves two purposes: availability and performance. An LDAP client can choose among several servers, increasing overall availability. Overall performance is increased by load-balancing client requests among multiple LDAP server systems (although LDAP does not include a mechanism for automatic load balancing).

The use of LDAP replica servers is always recommended when availability is a concern. Configuration of an LDAP replica server using the IBM Directory Server simply requires specifying some self-explanatory information via the administration interface. Before replication can work, however, a few prerequisites must be met:

- The LDAP replica server must have the same suffix(es) defined as the LDAP master server. Suffixes do not get replicated automatically.
- If the LDAP master server uses schema changes, the LDAP replica server must be updated with these changes before replication can take place. This is of importance in the context of this book as DCE and other components make use of schema changes.
- If an LDAP master server is already populated with directory information, that information must be manually transferred to the replica before replication can work.

7.7 SSL implementation hints

Secure Sockets Layer (SSL) Version 3 and Transport Layer Security (TLS) Version 1 are protocols that provide server authentication, client authentication (optional), and protection by means of privacy and data integrity between two communicating entities. Implementations of SSL and TLS are available on many platforms including AIX, OS/390, OS/400, Linux, Windows 2000, and Solaris.

7.7.1 SSL and TLS overview

SSL is a Netscape proprietary protocol that has become a de facto standard. TLS is an industry-standard protocol that is based on SSL and defined in IETF RFC 2246, "TLS Protocol Version 1.0." Both the SSL and TLS protocols use standards defined in IETF RFC 2459, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile." The two protocols have few differences and often are referred to interchangeably.

Several products recommended in this book, including IBM WebSphere Application Server, contain embedded SSL/TLS. Customers can use the embedded SSL/TLS to replace the DCE authentication and protection services. The use of SSL is recommended between systems that exchange sensitive data and/or when strong authentication is required. The major differences between SSL/TLS and the DCE authentication and protection services are:

- SSL/TLS operates at a transport level, whereas DCE authentication and protection operates at a higher level.
- When using SSL/TLS, server authentication is required and client authentication is optional. When using the DCE authentication service, client authentication is required and server authentication is optional.
- When using SSL/TLS, all of the data transmitted over the connection must be protected through encryption and integrity. When using DCE, the authentication service can be used independently of the protection service so authentication can be performed without data protection or with selected data protection.
- SSL/TLS uses the public key infrastructure (PKI) protocol for authentication. DCE uses the Kerberos protocol for authentication. (DCE has the option of using PKI for initial authentication, but supports only the Kerberos protocol for client and server authentication.)

The differences between SSL Version 3 and TLS Version 1 are only minor and not important for the subject of this book. Because the term *SSL* is more often used throughout the industry, it is used in this book as a synonym for both protocols.

7.7.2 Uses of SSL

SSL is used for encryption (privacy) and authentication. SSL uses Public Key Infrastructure (PKI) technology, namely X.509 digital certificates and digital signatures, to authenticate a server or a client, and the process of authentication is carried out by the SSL implementation (SSL libraries).

A server or a client that has to authenticate with SSL needs a *certificate* (more precisely, an X.509v3 digital certificate). It also needs a *private key* (sometimes also referred to as a secret key) that belongs to the certificate, because only the possession of the private key *and* the certificate is evidence of authenticity.

> **Note:** In common language, the phrase “authentication with a certificate” is, strictly speaking, incomplete and misleading. A certificate is, by definition, not confidential and maybe copied easily. Thus, the possession of a certificate cannot be proof of authenticity. Only the possession of the *private key* that belongs to a certificate is considered proof of authenticity. In contrast to certificates, private keys must not be shared and must be protected by all means. When a subject authenticates to another subject using SSL and certificates, a complex handshaking protocol takes place that includes digital signatures using the private key. The private key, however, is never transmitted over the network and hence this is strong authentication. The SSL handshake protocol is transparent to the user or the application.

7.7.3 Using SSL in the replacement scenarios

Most products mentioned in this book, including DCE, IBM Network Authentication Service, IBM Directory Server, IBM Tivoli Access Manager, and IBM WebSphere, support SSL for encryption and authentication when communicating with other products. Specifically:

- IBM Directory Server supports SSL for the protection of the communication with its clients.
- DCE, IBM Network Authentication Service, IBM Tivoli Access Manager, and IBM WebSphere Application Server all support SSL when communicating with IBM Directory Server.
- IBM WebSphere supports SSL for the CORBA application client to authenticate to the CORBA application server.
- The HTTP Server, as part of IBM WebSphere, supports SSL for Web browsers.

Although these products support SSL, and using SSL is recommended for security reasons wherever feasible, for the sake of simplicity the scenarios in the

following chapters do not usually use SSL. The only instances where SSL is used is for replacing secure RPC.

7.7.4 IBM GSKit

The IBM products as listed in the previous section all use a common tool, called the IBM GSKit (sometimes also referred to as the ikeyman tool), that constitutes an implementation of the SSL protocol. Besides the SSL protocol handling, IBM GSKit also supports elementary functions for certificate handling through a graphical administrator interface, such as:

- Creating self-signed certificates for temporary or lab-type installations
- Creating certificate requests for later approval by a Certificate Authority
- Storing personal certificates (including server certificates) in a certificate file
- Storing root certificates (also called signer certificates) in a certificate file

Figure 7-1 shows a simplified representation of the IBM GSKit. As can be seen, it manages the SSL protocol and contains two lists of certificates, the personal and the root certificates of trusted CAs. The graphical user interface is not shown.

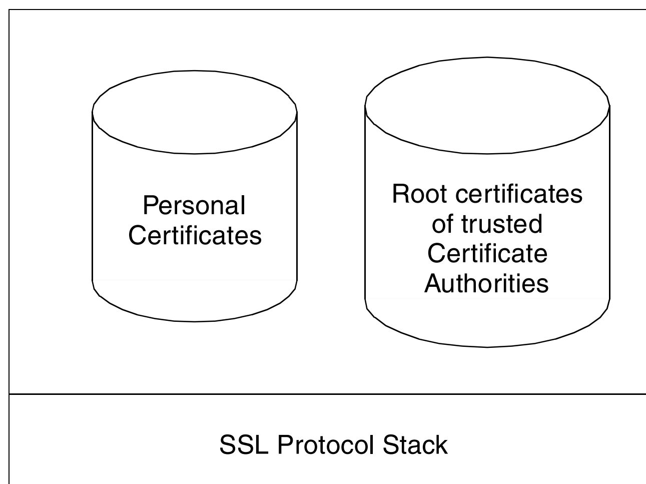


Figure 7-1 The components of IBM GSKit (simplified)

With these elementary functions, IBM GSKit serves as the primary administrator tool for managing certificates and setting up secure SSL connections.

IBM GSKit ships with the products that use and support it and is not available separately. Refer to the documentation that comes with the respective products for further information and specific instructions.

7.7.5 Authentication with certificates

A certificate, together with the private key, proves the authenticity of an entity such as a person or a system. (In PKI terminology, an entity is also called a subject.) A certificate provides evidence that, at the time of certificate issuance, the subject possesses the private key that belongs to the certificate. Assuming that the private key is not stolen, lost, or otherwise compromised, the subject is the only person or system that can use this private key together with the certificate to prove its authenticity.

The authentication process itself involves a number of substeps that are beyond the scope of this book to detail as many variants and exceptions as must be considered. They require a sound understanding of public key cryptography, including the handling of cases when the private key is lost or stolen.

Nevertheless, a few statements can be made that must be met in order to make certificate authentication work:

- The SSL implementation (such as IBM GSKit) has the technical means to ensure that a certificate belongs to a certain subject. This is an important function because a certificate could be copied and misused by fraudulent subjects.

  For example, an application server that receives a certificate with a DN of John Doe can rely on the SSL implementation that it is really John Doe who authenticated to that server. There is no need for a password or the like.

- In order for the SSL implementation to verify the validity of a subject's certificate, it must know and trust the authority that issued the certificate.

  Trusted *Certificate Authorities* (CA) issue *root certificates* required by an SSL implementation to verify the validity of a subject's certificate. The SSL implementation (IBM GSKit) must have a copy of the root certificate of the CA that issued the subject's certificate.

- For authorization and audit purposes, a subject's certificate must contain some information that can be used for such purposes.

  According to the X.509v3 standard, a certificate may contain various information. The *subject DN* is one piece of information that is contained in all certificates. Although the format of this text string is not strictly specified, it is obvious that the subject DN can be used as an identifier for the subject.

To sum up: A certificate must contain some information that can be used by the receiving party to identify the subject. The receiving partner also must have a copy of the root certificate of the CA that issued the certificate. The technical details of the authentication and encryption processes are hidden and carried out by the SSL implementation, such as the IBM GSKit.

The application programmer or system administrator must enable SSL in the application or product and provide necessary references to the respective IBM GSKit resources in order to use SSL. Before SSL can be used with IBM GSKit, however, the certificate files (also referred to as key files, trust files, or key databases) must be populated with the certificates to be used.

An administrator has different options for obtaining certificates, including:

- Self-signed certificates: GSKit allows generating self-signed certificates for temporary or testing purposes.
- Certificates from a Certificate Authority (CA): A CA is a trusted authority that can issue trusted certificates.

7.7.6 Using self-signed certificates

Using self-signed certificates simplifies the administration process by not involving a separate CA. Self-signed certificates can be used as regular certificates for authentication purposes (for people and systems), and, at the same time, they can be used as root certificates.

Figure 7-2 shows an example of two application partners, Joe and Sue, that require mutual SSL certificate authentication. Note that it is irrelevant in this example whether Joe or Sue are actually application servers or clients.

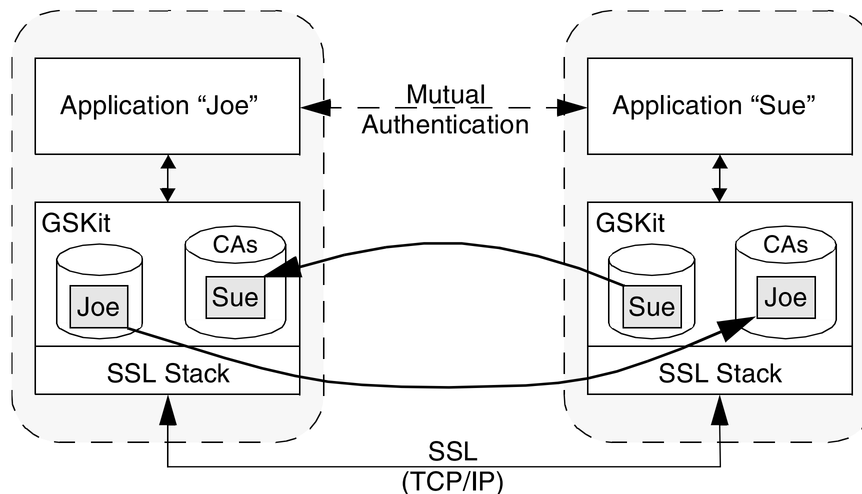


Figure 7-2 Using self-signed certificates

The following actions are required to make the example work.

The administrator:

1. Creates a self-signed certificate with IBM GSKit for application Joe.
2. Creates a self-signed certificate with IBM GSKit for application Sue.
3. Imports Joe's self-signed certificate (as a root certificate) into the list of trusted CAs on Sue's GSKit.
4. Imports Sue's self-signed certificate (as a root certificate) into the list of trusted CAs on Joe's GSKit.

The application administrator:

1. Configures the application to use the specific certificate file(s) created by the administrator.
2. Specifies which certificate from within the certificate file to use. (There can be multiple certificates in a certificate file.)
3. (Optionally) Specifies certain encryption algorithms (cipher specs) for the SSL encryption.

The application programmer:

1. Initializes and uses SSL for session initiation. (The details may vary largely depending on the application.)
2. After SSL session initiation on the receiving partner, extracts the required information from the partner's certificate. This is most likely the subject DN that will be extracted.
3. If required, looks up further information, such as group membership or authorization data of the subject, by calling additional services not provided by SSL and/or GSKit.

> **Note:** This example shows mutual authentication. When only server authentication is required, then only one self-signed certificate has to be created for the server and imported into the client's GSKit.

As an alternative to self-signed certificates, some products such as IBM WebSphere ship with ready-to-use certificates that allow for a quick start of the product(s). These certificates are shipped in keyfiles, so the administrator does not have to import or otherwise deal with them. For a production environment, however, certificates from a CA are recommended.

7.7.7 Using certificates from a Certificate Authority (CA)

Using certificates from a CA has some advantage over using self-signed certificates. In terms of security, more-stringent rules may be applied when a

trusted third party issues certificates. Also, only one root certificate has to be imported into all involved GSKits, rather than mutually importing self-signed certificates from each possible communication partner. The process, however, involves different procedures, as shown in Figure 7-3.

A trusted CA can be either a commercial third-party provider, such as VeriSign, Inc. (`http://www.verisign.com`), or a private CA run and managed by organizations for their own needs. The open source community runs a project called OpenSSL (`http://www.openssl.org`) that provides tools that may be used for running a private CA (subject to terms and conditions). Alternatively, products and services are available on the market for running private CAs.

In the example depicted in Figure 7-3, only one-way authentication is shown; Joe needs to authenticate to Sue. There is no change in the process for the application administrator and the application programmer, but the generation of the certificate differs from using self-signed certificates.

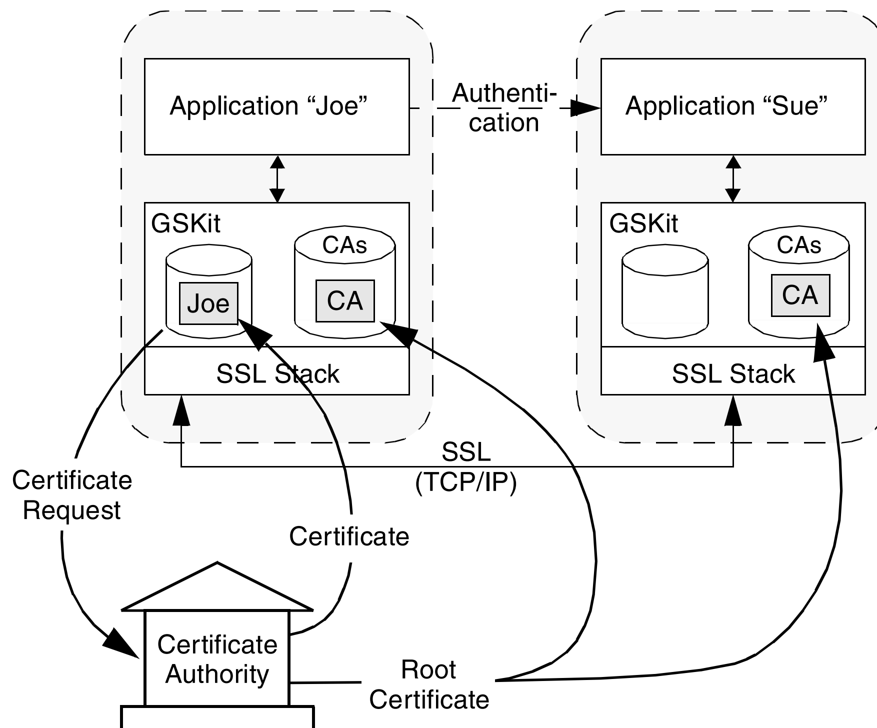


Figure 7-3 Using certificates from a Certificate Authority

In contrast to using self-signed certificates, the following must be done.

The administrator:

1. Imports the CA's root certificate into the list of trusted CAs on Joe's and Sue's GSKit. Most commercial CAs have their root certificates on their Web sites for download at no cost.
2. Creates a certificate request on Joe's GSKit and sends it to the trusted CA. This step involves the generation of the private key, which remains in the key file maintained by GSKit for Joe.

   The CA will then, based on its terms and conditions, generate and issue a certificate for Joe. (Remember that "Joe" may be a person or a system.)
3. Receives Joe's certificate from the CA and stores it in Joe's GSKit.

7.7.8 Additional hints and considerations

Although the process of certificate administration and SSL setup is straightforward and, in principle, the same for all products or platforms, a few considerations and pitfalls lead to questions. These include:

- Certificates, including root certificates, can be stored in various formats, including some that are not supported by, say, IBM GSKit. This could prevent the certificate from being imported. Try to generate the certificate in, or convert it to, a format that is supported by IBM GSKit.
- The key files managed by IBM GSKit also contain the private key(s) of the personal certificates. Because private keys should be treated as confidential, proper file and password protection for these key files is strongly recommended.
- By default, IBM GSKit comes with a list of root certificates of publicly known trusted CAs. For maximum security, administrators may consider reducing the list of root certificates to a minimum.
- For authorization and audit purposes, most applications extract the subject DN (or any other unique identifier) from the certificate. This often means that additional filters and/or code must be applied or written in order to match the certificate information to known identifiers.
- If a specific format or information has to be stored in a certificate, such as in the subject DN field, bear in mind that most tools or public CAs have some restrictions or rules that may prevent you from storing your specific information in the desired format.
- When using SSL, most products allow a wide range of encryption ciphers to be used. For maximum security, only strong ciphers should be allowed. This can be configured in most applications or products to override the defaults.

- All certificates have a validity period of two, five, or 10 years. After the validity period expires, the certificate will be rejected by most SSL implementation. Thus, a process must be in place to renew certificates before they expire. This is especially essential on production systems.
- Most commercial CAs also offer test certificates that can be used like normal certificates. Be aware that such test certificates often have only a short validity period of a few weeks or months and that a special root certificate for the test CA must be imported into IBM GSKit.
- On production systems, a root certificate of a test CA should never be installed in IBM GSKit. It would allow successful SSL connection (and maybe even authentication) of anyone who has a test certificate of that CA, which normally can easily be obtained.

8

Scenario 1: GSS-API application

This chapter presents scenario 1, which demonstrates C/C++ replacement strategies for an application that directly depends on the DCE login, GSS-API, authorization, and PAC services. The strategies are:

- Leave the entire application in C/C++.
- Replace DCE login and GSS-API with IBM Network Authentication Service and the Microsoft Windows implementation of the Kerberos client.
- Replace DCE PAC and authorization with IBM Tivoli Access Manager.
- Migrate the DCE registry database to IBM Directory Server, and configure IBM Network Authentication Service and IBM Tivoli Access Manager to use the migrated data.

Before reading this chapter, make sure that you read the previous Chapter 7, “Common replacement considerations” on page 111.

© Copyright IBM Corp. 2003. All rights reserved.

8.1 Scenario description

This scenario concerns an example application with DCE dependencies that is revised in order to remove the DCE dependencies. The chapter contents are:

- The initial application with its DCE dependencies is discussed first, followed by an overview of how the DCE dependencies are replaced.
- The application with DCE dependencies is detailed in section 8.2, “DCE application” on page 129.
- The replacement roadmap is explained in section 8.3, “Replacement roadmap” on page 138.
- The revised application, including coding fragments, is explained in section 8.4, “Revised application discussion” on page 152.
- The major considerations for the management of the new environment are discussed in section 8.5, “Administration considerations and interfaces” on page 157.
- The chapter concludes with section 8.6, “Discussion and conclusions” on page 168.

8.1.1 Initial application with DCE dependencies

This part of the scenario demonstrates a client/server application that depends on DCE for authentication, authorization, and protection of data. The application client is on the Windows platform and the application server is on the AIX platform.

The application logic is simple: The application client sends a character string to the application server, which reverses the character string and returns it to the application client.

Figure 8-1 on page 128 illustrates the DCE dependencies of the application. The application directly depends on the DCE login, GSS-API, PAC, and authorization services. The application indirectly depends on the DCE KDC, privilege, and registry servers, the DCE registry database, and an application-specific authorization database. The application also indirectly depends on the DCE directory server and namespace database, but this indirect dependency is only during replication of the registry database and is not shown in the illustration.

The application does the following:

1. The application client:
   a. Calls the DCE login API to obtain a security credential. The DCE login service processes this API by obtaining a ticket granting ticket (TGT) from

the DCE KDC server and a privilege TGT (PTGT) from the DCE privilege server. (A PTGT is a TGT with an embedded DCE PAC.)

   b. Calls the DCE GSS-API to obtain a service ticket (containing the same embedded DCE PAC) from the KDC server and wraps this service ticket in a GSS-API authentication token.

   c. Sends the GSS-API authentication token to the application server.

2. The application server then:

   a. Calls the DCE GSS-API to verify the client GSS-API authentication token and obtain a GSS-API mutual authentication token.

   b. Sends the GSS-API mutual authentication token to the application client.

3. The application client:

   a. Calls DCE GSS-API to verify the mutual authentication token.

   b. Calls DCE GSS-API to encrypt a message and wrap it in a GSS-API data token.

   c. Sends the GSS-API data token to the application server.

4. The application server:

   a. Calls DCE GSS-API to decrypt the GSS-API data token.

   b. Calls DCE GSS-API to get the DCE PAC from the client GSS-API authentication token.

   c. Calls DCE authorization API to get the client permissions from the client DCE PAC and an ACL stored in a simulated authorization database.

   d. Determines whether the client is authorized by comparing the client permissions with the desired permissions.

   e. If the client is authorized, reverses the string in the client message and sends the reversed string to the application client.

Note: The application server in this example does not perform a DCE or Kerberos login. Using Kerberos security (DCE security is based on it), only the context initiator has to obtain a Kerberos TGT, and therefore perform a Kerberos login. The application server in this scenario is a context acceptor only. However, some DCE application servers would still need to perform a DCE or Kerberos login, because they become clients themselves to other servers, such as DCE CDS.

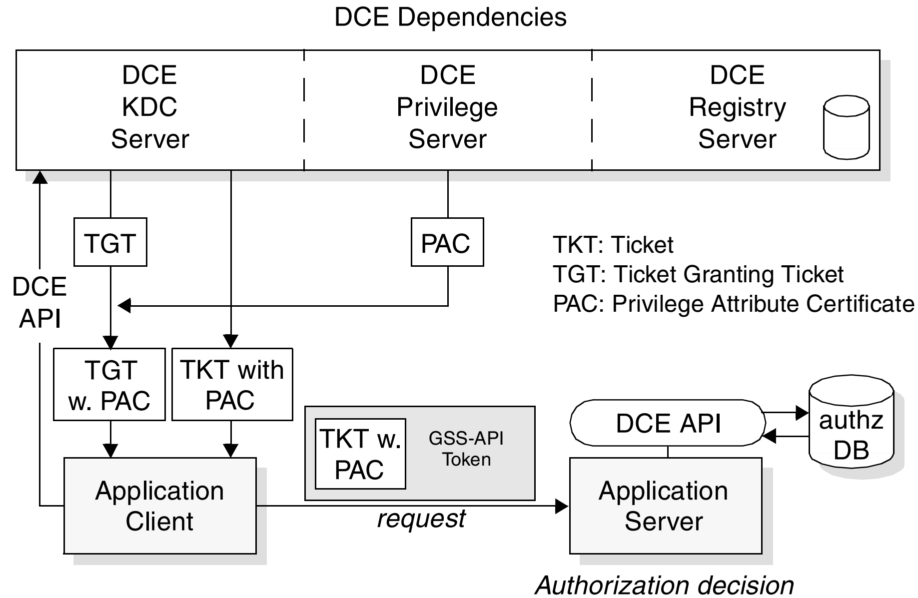


Figure 8-1 Authentication and authorization with DCE dependencies

8.1.2 Revised application without DCE dependencies

Figure 8-2 on page 129 shows the revised application. The DCE direct dependencies are replaced as follows:

- On the application client, the DCE login and GSS-API APIs are replaced with Windows APIs and the Windows login utility. Note that Windows requires some configuration in order to use a remote KDC service.
- On the application server, the DCE GSS-API APIs are replaced with IBM Network Authentication Service GSS-API APIs, and the DCE authorization APIs are replaced with IBM Tivoli Access Manager APIs (also referred to as *aznAPI*).

The DCE indirect dependencies are replaced as follows:

- The DCE registry server is replaced with the IBM Directory Server, and the DCE registry database is migrated to this IBM Directory Server (representing an LDAP directory).
- The DCE KDC server is replaced with the IBM Network Authentication Service KDC server, and this server is configured to use the migrated registry database.

- The DCE privilege server is replaced with the IBM Tivoli Access Manager policy server, which is configured to use the migrated registry database.
- The application-specific authorization database is migrated to the IBM Tivoli Access Manager policy database.
- The authorization logic of the application server is replaced with the IBM Tivoli Access Manager authorization database and according functions.

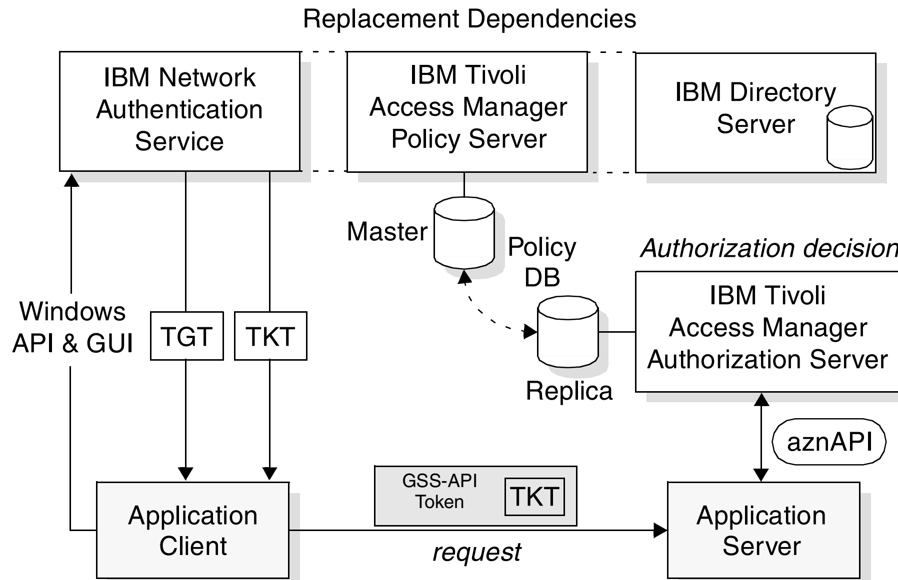


Figure 8-2 Authentication and authorization without DCE dependencies

Note that, depending on the setup, the IBM Tivoli Access Manager authorization database may be replicated to the system where the application server resides. Planning, performance and installation considerations for the IBM Tivoli Access Manager, however, are beyond the scope of this book.

8.2 DCE application

This section briefly describes the sample program source code, which makes use of DCE services to secure its communication. The DCE dependencies of this program are replaced as described in the subsequent sections of this chapter. The key DCE services that are used by this program are extracted and explained separately. The use of input and output parameters of the individual API calls are described in the order they appear in the respective service calls. The communication mechanism between client and server is TCP/IP sockets.

The application logic exclusive of the security programming consists of the application server returning a string, delivered by the client, in reversed byte order. For the sake of simplicity, only the application messages (strings) that the application client sends to the application server are protected by GSS-API. The reply of the server takes place in clear text. The complete source code can be found in Appendix B, “Scenario 2: Source code listings” on page 331.

8.2.1 Configuring and running the DCE application

For the sample program to run, the following principals and groups have been entered into the DCE security registry:

scenario1_princ — The principal used by the application client to acquire network credentials.

scenario1_server_princ — The principal used by the application server to acquire network credentials.

scenario1_clients — A group, which is part of the ACL of the resource to be protected. Both *scenario1_princ* and *scenario1_server_princ* are members of this group.

scenario1_resource — The actual resource to be protected. This is not used in the DCE example, as the application holds its resources in its own, simulated data store.

These registry objects are later part of the migration of DCE registry objects to the LDAP directory and the IBM Tivoli Access Manager.

Start the applications by entering their executable names on the command line. Note that changes likely will be necessary to the hard-coded server host name in the client example to adapt the program to your environment.

8.2.2 Application client

Upon startup, the application client in this DCE-dependent example logs in to DCE, using the DCE security login API. Subsequently, the application client initiates a security context with the application server. If this security context is accepted by the server, the client starts sending encrypted application data to the application server. Example 8-1 shows the typical DCE login code. In the process of this login, a DCE PAC is added to the security ticket. The DCE PAC is added through a call to the DCE privilege server, made transparently for the application.

Example 8-1 DCE login code

```
int login_to_dce(
   unsigned_char_p_t prin_name,   /* Server principal name.*/
   unsigned_char_p_t keytab)      /* Keytab file */
```

```
{

   sec_passwd_rec_t         *keydata;
   sec_login_auth_src_t     auth_src;
   boolean32                reset_pwd;
   sec_login_handle_t       login_context;
   unsigned32               status = error_status_ok;

   /***** Create a context and get the login context *****/
   sec_login_setup_identity(
       prin_name,
       sec_login_no_flags,
       &login_context,
       &status);
   if (status != error_status_ok){
      fprintf(stderr,
         "Error - sec_login_setup_identity(), status: %d\n",status);
      return -1;
   }

   /***** Get secret key from the keytab file *****/
   sec_key_mgmt_get_key(
      rpc_c_authn_dce_secret,
      keytab,
      prin_name,
      0,
      (void**)&keydata,
      &status);
   if (status != error_status_ok){
      fprintf(stderr,
         "Error - sec_key_mgmt_get_key(), status: %d\n",status);
      return -1;
   }

   /***** Validate the login context *****/
   sec_login_validate_identity(
      login_context,
      keydata,
      &reset_pwd,
      &auth_src,
      &status);
   if (status != error_status_ok){
      fprintf(stderr,
         "Error - sec_login_validate_identity(), status: %d\n",status);
      return -1;
   }

   /***** Finally, set the context *****/
   sec_login_set_context(
```

```
      login_context,
      &status);
   if (status != error_status_ok){
      fprintf(stderr, "Error - sec_login_set_context(), status: %d\n",status);
      return -1;
   }

   printf("DCE login successful.\n");
   return 0;
}
```

Example 8-2 on page 133 shows the use of the GSS-API function gss_init_sec_context to create an authentication token, which the application client then sends to the application server in order to prove its identity and to deliver a session key to the server. If mutual authentication is required, which is the case in this example, the application server also is required to send a token back to the client. Then, the application client can verify this token to authenticate the application server. After successful completion of this mutual authentication process, the application client and server share a common secret key for further communication.

The following is a brief description of the parameters as they are used in the gss_init_sec_context API call: After the usual status parameter (min_stat), GSS_C_NO_CREDENTIAL indicates that the client assumes default credentials. Out parameter sec_ctx takes the created security context. In server, the name of the context acceptor is passed in. In this example it is the principal name, which the server program uses. GSSDCE_C_OID_DCE_KRBV5_DES specifies the security context to be used. The security context in this example supports three service options: Mutual authentication, data integrity, and data encryption. This is achieved by the bitwise Ored flags GSS_C_MUTUAL_FLAG, GSS_C_INTEG_FLAG and GSS_C_CONF_FLAG. 24*60*60 specifies the desired period of time in seconds for which the context should remain valid. In this example the period shall be 24 hours. No channel binding should be used, which is indicated by GSS_C_NO_CHANNEL_BINDINGS. Channel bindings enable the GSS-API callers to bind the establishment of the security context to relevant characteristics such as addresses or to application-specific data.

Note that because in this example mutual authentication is requested, gss_init_sec_context is actually called in a loop. That means that the client initiates a security context and sends an authentication token (snd_token) to the server, which in turn sends an authentication token back after it has accepted the client's context. The server's token (rcv_token) is then used by the client to call gss_init_sec_context one more time. This way, the client authenticates the server. The three NULL output parameters specify that the level of protection, its quality, and the actual context lifetime are not requested to be returned.

Example 8-2 GSS setup of initial security context

```
maj_stat = gss_init_sec_context(
   &min_stat,
   GSS_C_NO_CREDENTIAL, /* Default cred */
   sec_ctx,
   server,
   GSSDCE_C_OID_DCE_KRBV5_DES,
   GSS_C_MUTUAL_FLAG | GSS_C_INTEG_FLAG | GSS_C_CONF_FLAG,
   24*60*60,
   GSS_C_NO_CHANNEL_BINDINGS,
   &rcv_token,
   NULL,
   &snd_token,
   NULL,
   NULL);
```

Example 8-3 shows the use of the GSS-API functions gss_seal and gss_unseal. These functions are used for securing the exchange of data between the application client and the application server with regards to both the data type conversions between different platforms and data protection. In newer versions of GSS-API, this functions have been replaced by gss_wrap and gss_unwrap.

With gss_seal, after the status and security context parameter, the parameter true indicates that both integrity and confidentiality (encryption) services are requested. GSS_C_QOP_DEFAULT specifies the cryptographic algorithm (quality of protection). The DCE GSS-API only supports one quality of protection. Next, the actual application message is supplied as input in message. Of the 2 remaining output parameters, NULL indicates that the level of protection is not requested in this case and token is the application message, sealed (encrypted) and returned in a GSS token for later transmission to the server.

With gss_unseal status and the security context are specified again as the first two parameters. In the next two parameters, the GSS token (token) is unpacked into the decrypted application message (message). The level of protection and its quality again are not requested here, indicated by the last two NULL parameters.

Example 8-3 GSS wrapping and unwrapping of messages

```
maj_stat = gss_seal(
   &min_stat,
   security_context,
   true,
   GSS_C_QOP_DEFAULT,
   &message,
   NULL,
   &token);
```

```
maj_stat = gss_unseal(
   &min_stat,
   security_context,
   &token,
   &message,
   NULL,
   NULL);
```

Note: The preceding and following examples often use the value NULL instead of an output parameter. This is common practice when an output is not requested.

8.2.3 Application server

At startup, the application server in this example registers its identity with the DCE run-time environment. Then it acquires the credentials for the registered identity, which it subsequently uses to accept the security context initiated by the client. After mutual authentication is achieved successfully, the actual exchange of application messages between the application server and the application client can take place.

In this example, the application client sends a line of text to the server, which the server simply returns in reversed character order. All messages are encrypted and packed in GSS tokens for communication. Additionally, the server contains authorization code, based on DCE ACLs, to check the client's permissions for each message sent.

Example 8-4 shows the use of gssdce_register_acceptor_identity. This DCE GSS API registers the application server with the DCE run time, which includes registration of the server principal name and the location of the keytab file to be used. The NULL parameter indicates that no special function for key-retrieval shall be used.

Example 8-4 GSSDCE register acceptor identity

```
gssdce_register_acceptor_identity(
   &min_stat,
   server_name_t,
   NULL,
   file_param);
```

In Example 8-5 on page 135, the server acquires credentials for its principal name and key.

The period of time in seconds for which the credentials should be valid is specified by 24*60*60. The example uses GSS_C_NULL_OID_SET to specify the default security mechanism. GSS_C_ACCEPT tells the run time that these credentials should only be used to accept a security context, rather than initiating one. This way, as mentioned earlier, no Kerberos or DCE login is required by the server. Out parameter cred_handle is the handle to the created credentials. The actual mechanism and credential lifetime are not of interest here, which is indicated by the last two NULL parameters.

Example 8-5 GSS acquire credentials

```
maj_stat = gss_acquire_cred(
   &min_stat,
   server_name_t,
   24*60*60,
   GSS_C_NULL_OID_SET,
   GSS_C_ACCEPT,
   &cred_handle,
   NULL,
   NULL);
```

In Example 8-6, gss_accept_sec_context corresponds to the API gss_init_sec_context, which the client program uses as shown in Example 8-2 on page 133. This API allows the server program to verify the ticket received from the client in order to authenticate the client and to extract a session key.

The sec_ctx parameter is the handle to the security context that was created after the function call succeeded. The handle to the server credentials cred_handle, obtained in the pervious call to gss_acquire_cred, is passed in. Like on the client side, no channel binding should be used, which is indicated by GSS_C_NO_CHANNEL_BINDINGS. Channel bindings enable the GSS-API callers to bind the establishment of the security context to relevant characteristics such as addresses or to application-specific data. Given this input, together with the token rcv_token received form the client, a token snd_token is created, which is then sent back to the client. snd_token will be verified by the client to authenticate the server for mutual authentication. As additional information, the principal name of the client is returned in client.

Example 8-6 GSS accept security context

```
maj_stat = gss_accept_sec_context(
   &min_stat,
   sec_ctx,
   cred_handle,
   &rcv_token,
   GSS_C_NO_CHANNEL_BINDINGS,
   &client,
```

```
   NULL,
   &snd_token,
   NULL,
   NULL,
   NULL);
```

Authorization code

As mentioned previously, the DCE example makes use of DCE-dependent authorization code based on DCE ACLs. It is assumed that the application maintains its own ACLs in application-specific data stores. Usually this also involves the use of a DCE ACL manager, using the DCE *rdacl* interface. However, as rdacl is not involved in the actual authorization decision, it is not included in this example.

The example shows how the core of ACL-based authorization works, without using direct RPC and CDS dependencies. This explains which DCE authorization code has to be revised in order to remove DCE dependencies. Example 8-7 shows how authorization is achieved in the scenario example. The complete source code can be found in “Authorization module with DCE dependencies” on page 281.

Example 8-7   DCE authorization code

```
boolean32 is_client_authorized(
    gss_ctx_id_t        security_context,
    sec_acl_permset_t   desired_perms){

    rpc_authz_cred_handle_t          cred_h;
    sec_acl_t                        *sample_acl;
    rpc_authz_cred_handle_t          output_cred;
    OM_uint32                        maj_stat, min_stat;
    error_status_t                   status;
    sec_acl_permset_t                perms  = dce_acl_c_no_permissions;

    /* Set up sample a ACL                       */
    setup_sample_acl(&sample_acl, &status);
    if (status != error_status_ok){
       fprintf(stderr,
          "Error - setup_sample_acl(), status: %d\n",status);
       return false;
    }

    maj_stat  = gssdce_extract_creds_from_sec_context(
                  &min_stat,
                  security_context,
                  &output_cred);
    if (GSS_ERROR(maj_stat)) {
```

```
        display_error(
           "gssdce_extract_creds_from_sec_context", maj_stat, min_stat);
        exit (EXIT_FAILURE);
    };

    dce_acl_inq_permset_for_creds(
       output_cred,
       sample_acl,
       NULL,
       NULL,
       sec_acl_posix_no_semantics,
       &perms,
       &status);

    if (status != error_status_ok){
       fprintf(stderr,
          "Error - dce_acl_inq_permset_for_creds(), status: %d\n",status);
       return false;
    }

printf("granted perms: %x\n",perms);

    /* If desired permissions are included in caller permissions,
       caller is authorized */
    if ( (desired_perms & perms) == desired_perms ){
        printf("access granted.\n");
        return true;
    }

    return false;
}
```

The function is_client_authorized in Example 8-7 on page 136 uses the following functions and DCE APIs to achieve authorization:

1. setup_sample_acl

   This function simulates the use of an application-specific ACL store. The ACL to test against is fetched from this store.

2. gssdce_extract_creds_from_sec_context

   This DCE API pulls the DCE PAC information from the GSS security context.

3. dce_acl_inq_permset_for_creds

   This DCE API compares the client's PAC information with the ACL read from the applications data store, so the list of access rights is returned as a bitmap.

The authorization decision is then made by testing this bitmap against the desired access rights.

8.3 Replacement roadmap

This section describes the migration path for the DCE application. The environment must be migrated in the order shown in the following migration roadmap, which is explained in the upcoming sections:

1. Fulfill the software requirements.
2. Configure a DCE security replica server to store its security registry in a shared LDAP directory (IBM Directory Server).
3. Configure the IBM Network Authentication Service (Kerberos) for authentication using the data in the LDAP directory.
4. Configure the IBM Tivoli Access Manager for authorization using the data in the LDAP directory.
5. Configure the Windows client(s) to use a remote Kerberos authentication service.
6. Revise the application(s).
7. Remove DCE or DCE dependencies.

8.3.1 Software requirements

This scenario uses the following software:

- IBM Directory Server Version 3.2.2 with e-fix SWD-002

  E-fixes are available at:

  `http://www.ibm.com/software/network/directory/support/fixes`

  > **Note:** The former name of the IBM Directory Server at Version 3.2.2 was *IBM SecureWay Directory*. It will be referred to as the IBM Directory Server.

- IBM Network Authentication Service V1.3

  Install this using the AIX Version 5.1 5L Expansion Pack.

- IBM Tivoli Access Manager V3.9, namely its following components:
  - Access Manager Runtime (PD.RTE)
  - Access Manager Authorization Server (PD.Acld)
  - Access Manager Policy Server (PD.Mgr)
  - Access Manager Application Developer Kit, if required (PD.AuthADK)

- Microsoft Platform SDK - Core

  This can be downloaded from:

  `http://www.microsoft.com/msdownload/platformsdk/sdkupdate/default.htm`

- Microsoft Client Configuration to a non-Windows KDC
  - Windows 2000 Resource Kit Tool
  - Windows 2000 Support Tool - Provided on the Windows 2000 CD-ROM

8.3.2 Migration of DCE security registry to IBM Directory Server

This section briefly describes the procedure to migrate the DCE security registry to IBM Directory Server. It is important to understand that only a DCE security replica server is used for this migration as the first step. This way, both the original DCE master security service and a copy of its registry in the LDAP directory are available at the same time during the migration time period. During that time period, the DCE master security server continues to accept updates and it keeps its registry in the local file system. This ensures continuous operation of the existing DCE cell.

For more detailed information, refer also to the *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*. (See “Online resources” on page 422.) The following is a contraction of this integration guide, provided here for your convenience.

Before Migration

To ensure a successful migration to utilize an LDAP directory as a security registry, verify that the following actions have been performed before beginning the migration:

- IBM DCE V3.2 is installed on all server machines that are going to be migrated. This is required because only Version 3.2 supports the migration of the registry to an LDAP directory.
- It is recommended that there are at least two DCE security replica servers in the DCE cell that you are migrating. This ensures best safety and availability should any error occur during the migration.
- The LDAP client package must be installed on all DCE security server machines that you are migrating.
- The master key must be available to each security server you are migrating. All DCE security servers using the LDAP directory must have access to the same master key file. If you store the master key in LDAP (by specifying the -master_key_in_ldap flag when you migrate), it is automatically available to all security servers. However, if you prefer to store the master key in a file, you must copy the key file to each server. The default location is the local file system in /opt/dcelocal/var/security/.mkey. Prior to migrating a legacy security server to LDAP, make a backup copy of the existing .mkey file on the server and then copy the common master key file to the local file system. To specify an alternate location for the key file, use the -dce_master_key flag when you

migrate the server. Ensure that the master key is secure. The legacy DCE implementation sets the permissions on the master key so it is accessible as read/write by root only.

- The DCE schema definitions for LDAP must be loaded into the LDAP directory. IBM DCE Version 3.2 ships with the following LDIF files for the IBM Directory Server Version 3.2.2 (additional LDIF files are also included for Netscape directory servers):
  - IBM.DCE.schema.ldif
  - IBM.KRB.schema.ldif

  These files contain DCE schema definitions and Kerberos schema definitions, respectively. These schema definitions are necessary to store DCE security data in the LDAP directory. The Kerberos portion of the schema might already be installed, but loading it again does not cause any problems. If the schema on the LDAP servers does not already have the DCE and Kerberos schema installed, you must install and load it on all of the LDAP directories that will be storing DCE security data.

  > **Important:** You must load the IBM Tivoli Access Manager schema files *before* you load the DCE schema.

  Use this command to load the schema files into the IBM Directory Server:

  ```
  ldapadd -c -D <bind DN> -w <passwd> -f <filename>
  ```

  Where bind DN is the DN of the administrator or an account that has sufficient write authority (for example cn=root), passwd is the password for this account and filename is one of the two LDIF files mentioned before. Run this command twice, first loading the Kerberos schema, then the DCE schema.

- The krbRealm for the DCE cell must be created in LDAP. You must configure an entry to represent the DCE security registry database. This entry is referred to as the *realm entry*. DCE uses this entry as the base for storing DCE security registry data. Do the following to configure the realm entry:
  - Choose a location for the realm entry in the directory information tree (DIT). This location must be serviced by trusted LDAP servers only.
  - Assign the realm entry an RDN of krbRealmName-v2=realmname where *realmname* is the name of the DCE cell without the preceding */.../*. For example, if the name of the DCE cell is /.../artichoke_cell, use artichoke_cell as the realm name.
  - Use the KrbRealm-v2 structural object class and the DCERealm auxiliary object class to configure the realm entry.
  - Add the required krbRealmName-v2 attribute to the entry, and configure this attribute with the same realm name that is in the RDN.

– Add the required krbPrincSubtree attribute to the realm entry, and store in this attribute a list of one or more DNs. Each DN represents a directory subtree entry under which DCE searches for DCE principal, DCE group, and DCE organization data. Do the following to ensure the security of each DN specified in the krbPrincSubtree:
  - Ensure that each DN resides in a directory location that can be serviced only by trusted LDAP servers.
  - Ensure that each DN can be protected with LDAP ACLs so that only trusted identities can configure DCE attributes under the DN.

  Be sure that at least one DN in the krbPrincSubtree entry is the DN of the realm entry itself or a DN under which the realm entry resides. Because LDAP performs searches fastest when able to search an entire suffix, it is recommended that you configure only the DNs of LDAP suffixes in the realm entry. However, for security reasons you should not configure the DN of a suffix in the krbPrincSubtree entry if any entries under the suffix are serviced by untrusted LDAP servers. You also should not configure the DN of a suffix in the krbPrincSubtree entry if you cannot protect all entries residing under the suffix from untrusted identities configuring DCE attributes.

– Add the krbKdcServiceObject attribute to the realm entry, and configure this attribute with the DN of each DCE security service entry.

– Add the krbTrustedAdmObject attribute to the realm entry, and configure this attribute with the DN of each DCE security service entry.

In the following example of an LDIF file, the DN of the realm entry is krbRealmName-v2=realm1,ou=Austin. This DN and the DN of cn=dept1,ou=Austin are the only DNs configured in krbPrincSubtree. Therefore, DCE searches only under the realm entry and the cn=dept1,ou=Austin for DCE principals, groups, and organizations. The realm contains two DCE servers: servera.deptabc.ibm.com and serverb.deptabc.ibm.com.

```
DN: krbRealmName-v2=realm1, ou=Austin
objectclass: KrbRealm-v2
objectclass: KrbRealmExt
krbRealmName-v2: realm1
krbPrincSubtree: krbRealmName-v2=realm1, ou=Austin
krbPrincSubtree: cn=dept1, ou=Austin
krbKdcServiceObject:
 serviceName=servera.deptabc.ibm.com,dc=deptabc,dc=IBM,dc=COM
krbKdcServiceObject:
 serviceName=serverb.deptabc.ibm.com,dc=deptabc,dc=IBM,dc=COM
krbTrustedAdmObject:
 serviceName=servera.deptabc.ibm.com,dc=deptabc,dc=IBM,dc=COM
krbTrustedAdmObject:
```

```
serviceName=serverb.deptabc.ibm.com,dc=deptabc,dc=IBM,dc=COM
```

- The LDAP ACLs must be set on the DCERealm and KrbMstrKey objects.
- A suitable LDAP bind protocol and authentication method must be selected. It is recommended to use SSL to secure the communication between the DCE security server and the IBM Directory Server. If not using SSL, wiretapping could be used to copy sensitive information as it transmits over the network.
- There must be one or more entries configured in LDAP to represent the DCE security server(s) in the realm. The DCE security servers use these entries to bind to LDAP. The following example represents such an entry:

```
dn: serviceName=servera,dc=itso,dc=ibm,dc=com
objectclass: cimManagedSystemElement
objectclass: eService
objectclass: cimManagedElement
objectclass: cimLogicalElement
objectclass: top
objectclass: KrbAuthObject
userpassword: {iMASK}>16rS7rRd7bSFdYPJ7GXi32d/1t+xE7sSyTHGI7QaiWcV1lzNNCnnZ
 qUov0QNOR7aRCTbppVUpQDMx3t2sN9OskGtjI7YCSxUvq3hqF8Ok0ha4O6kJnXgJVUgSBwcnVd
 vTge1zdkegoumOnfW9KcOacEUcDUOz<
servicename: servera
```

- The SEC_REP_INIT_FROM_UUID environment variable must be set on the master security server machine with the UUID of a legacy replica that will initialize the LDAP migration server. This enables the master server to continue accepting updates during initialization.

The steps as mentioned before prepare the environment such that the actual migration of a DCE security replica server to utilize the LDAP directory can take place, as described next.

Actual Migration

Perform the following steps to migrate a DCE replica security server to use an LDAP directory as its registry back end store:

1. Use the **dcecp** command to convert one legacy replica server into an LDAP migration security server:

```
dcecp -c registry migrate -migrationslave \
      -ldap_host <hostname:port | "list of hostnames"> \
      -bind_dn <bind_dn> -bind_dn_pw <bind_dn_pw> \
     [-ssl {-auth_type <ssl | cram-md5>} | \
            -auth_type <simple | ssl | cram-md5>] \
     [-keyring <ldap_keyring_file>] \
     [-keyring_pw <ldap_keyring_pw>] \
     [-master_key_in_ldap ] \
```

```
    [-dce_master_key <dce_master_key_file>] \
    [-delete_type <all | krb_dce | dce>]
```

After you perform this step, the migration server migrates all of the registry information to the LDAP directory and then uses this LDAP directory as the only backend store. The DCE cell can continue to operate in this state for an indefinite period of time. However, it is recommended that you migrate the security servers to LDAP once you are comfortable with the performance and stability of the cell. If you do not complete the migration of all of the security servers, any changes made to the LDAP directory using LDAP interfaces are not passed to the DCE master or any other replicas. After you are sure that the security data stored in LDAP is correct and is in sync with the legacy master's security data, move on to the next step.

If there is a problem at this stage, unconfigure the migration server and remove any extraneous data from LDAP, perform problem determination, and then start over.

2. Use the **dcecp -c registry migrate -ldapslave** command to convert all legacy replica servers into LDAP security slaves. After you convert the legacy replica servers to LDAP, they no longer have a local database or process the propagations from the legacy master. When they receive requests from clients, they retrieve the data from LDAP. Although it is strongly recommended that you continue the migration process, it is not mandatory. The DCE cell can remain in this state with a migration server and a legacy master for an undetermined amount of time.
3. Unconfigure all DCE legacy security servers that have not been migrated to LDAP.
4. Use the **dcecp -c registry modify -version secd.dce.1.3** command to raise the registry version to secd.dce.1.3. This only works if the legacy master server is running IBM DCE V3.2. To ensure that all downlevel servers shut down, issue any command that updates the registry database. When the master attempts to propagate the update, all downlevel servers shut down. Although this ensures that all downlevel servers shut down, it does not ensure that all level 3.2 security servers have been migrated to LDAP.

   Note: The registry version can be raised only one level at a time. If it is currently at 1.2.2, it must be raised to 1.2.2a before it is raised to 1.3.

5. Use the **dceback** command to back up legacy DCE security data on the legacy master. To do this, enter the following at the command prompt:

```
dceback dumpsecurity -destfile <sec_backup_file>
dceback dumpmisc -destfile <misc_backup_file>
```

6. Use the **dcecp -c registry migrate -ldapmaster** command to convert the legacy master into an LDAP security master.

> **Note:** After you convert the DCE legacy master to an LDAP security master, there is no way to revert to the legacy security function.

After you perform the last step, the LDAP security master writes security data to LDAP (rather than to memory and local databases), and the migration server becomes an LDAP security slave. Any DCE security servers that have not been migrated to LDAP no longer receive updates from the master and must be unconfigured. Replicas that do not receive updates return outdated information to clients. The DCE legacy master can be converted to an LDAP security master even if all of the DCE legacy slaves in the cell have not been converted to LDAP security slaves. If the promotion to the DCE security master fails, run the following commands:

```
stop.dce
dceback restoremisc -sourcefile <misc_backup_file>
dceback restoresecurity -sourcefile <sec_backup_file>
start.dce
```

8.3.3 Configuring IBM Network Authentication Service

Two documents that are included with the AIX product describe IBM Network Authentication Service in more detail. They are:

- *IBM Network Authentication Service Version 1.3 for AIX, Administrator's and User's Guide*
- *IBM Network Authentication Service Version 1.3 for AIX, Application Development Reference*

For details about how the IBM Network Authentication Service uses the DCE authentication data in the IBM Directory Server, refer to Chapter 4, "Using DCE data with IBM Network Authentication Service" on page 75.

For your convenience, the following is a brief description of what needs to be done. The configuration of the IBM Network Authentication Service to use the DCE authentication data that have been migrated to the IBM Directory Server can be done with one single command:

```
config.krb5 -h | -S [-a <admin>] \
            -d domain \
            -r realm \
            -s server \
          [[-l {ldapserver | ldapserver:port}] \
           [-u ldap_DN -p ldap_DN_PW] \
```

```
        [-f {keyring | keyring:entry_dn} -k keyring_pw] \
        [-m masterkey_location] \
        [-b bind_type] \
        [-R ldap_replica_list]]
```

For example, if your domain was deptabc.ibm.com and the realm was realm1, the LDAP security master was on host servera.deptabc.ibm.com (no SSL used), and the replica was on serverb.deptabc.ibm.com, then use the command:

```
config.krb5 -S -d deptabc.ibm.com -r realm1 -l servera.deptabc.ibm.com \
          -u cn=root -p Xc5dF3Kw -b simple -R "serverb.deptabc.ibm.com"
```

8.3.4 Configuring IBM Tivoli Access Manager

Configuring the IBM Tivoli Access Manager to use the migrated privilege data stored in the LDAP directory (IBM Directory Server) is described in Chapter 5, "Using DCE objects with IBM Tivoli Access Manager" on page 85. More information about the IBM Tivoli Access Manager can be found in the *IBM Tivoli Access Manager Base Administrator's Guide*. (See "Online resources" on page 422.)The following summarizes the steps involved:

1. Attaching the users and groups in the LDAP directory to the IBM Tivoli Access Manager.

   DCE users and groups must be attached to the information in the LDAP directory so that the IBM Tivoli Access Manager can use them. Rather than doing this manually, this section shows example UNIX shell scripts that perform this task more conveniently for an administrator, as most installations involve a large number of users and groups.

   Example 8-8 is a sample script to attach DCE users in the IBM Directory Server to the IBM Tivoli Access Manager information tree. It assumes that all the users are directly under the myapp directory tree and have unique names. The script uses a bind DN of cn=root and a password for this bind DN of Xc5dF3Kw. While writing scripts, care must be taken to preserve the directory structure of the principals.

Example 8-8 Attach users in IBM Directory to Access Manager

```
#!/bin/ksh

file1="principals"
file2="principals_new"

# Search the IBM Directory to find all the applciation users under the myapp
# directory tree
ldapsearch -D cn=root -w Xc5dF3Kw \
-b cn=myapp,cn=principal,krbRealmName-v2=realm1,dc=itso,dc=ibm,dc=com \
"(krbPrincipalName=*@realm1)" objectclass > $file1
```

```
# Modify the file to add the inetOrgPerson objectclass for each user.
cat $file1 | while read var
  do
    if [[ -n $var ]] then
      echo $var >> $file2
    else
      echo "objectclass=inetOrgPerson\n" >> $file2
    fi
  done

# Modify the IBM Directory entries with the inetOrgPerson objectclass added to
# all users
# under /myapp
ldapmodify -D cn=root -w Xc5dF3Kw -f $file2

# Retrieve the DN of all the DCE users
ldapsearch -D cn=root -w Xc5dF3Kw \
-b cn=myapp,cn=principal,krbRealmName-v2=realm1,dc=itso,dc=ibm,dc=com \
"(krbPrincipalName=*@realm1)" dn > $file1

# Import each DCE user (extract the username from the DN) into IBM Tivoli AM
and modify the user to activate the
# account. Any other modifications to the user can be done here.
cat $file1 | while read dn
  do
    if [[ -n $dn ]] then
      userName=$(echo $dn |sed -e 's/cn=//'|sed -e 's/,.*//')
      pdadmin -a sec_master -p Ye7Cu2ve user import $userName "$dn"
      pdadmin -a sec_master -p Ye7Cu2ve user modify $userName account-valid yes
      pdadmin -a sec_master -p Ye7Cu2ve user show $userName
    fi
  done

rm $file1 $file2
```

Example 8-9 is a sample script to attach DCE groups in the IBM Directory Server to the IBM Tivoli Access Manager information tree. Again, it assumes that all the groups are directly under the myapp directory tree and have unique names.

Example 8-9 Attach groups in IBM Directory to Access Manager

```
#!/bin/ksh

file1="principals"

#Search the IBM Directory for all the DCE groups and retrieve their DNs
ldapsearch -D cn=root -w Xc5dF3Kw \
```

```
        -b cn=myapp,cn=group,krbRealmName-v2=realm1,dc=itso,dc=ibm,dc=com \
        "(dceGroupName=*@realm1)" dn > $file1

# Import each DCE group (extract the group name from the DN) into the Access
# Manager
cat $file1 | while read dn
  do
    if [[ -n $dn ]] then
      echo $dn
      groupName=$(echo $dn |sed -e 's/cn=//'|sed -e 's/,.*//')
      echo $groupName
      pdadmin -a sec_master -p Ye7Cu2ve group import $groupName "$dn"
      pdadmin -a sec_master -p Ye7Cu2ve group show $groupName
    fi
  done

rm $file1
```

Note that the group membership attribute is shared between DCE and IBM Tivoli Access Manager. When a member is added to a group with DCE, that membership is also seen by IBM Tivoli Access Manager.

2. Creating an objectspace in the IBM Tivoli Access Manager and adding the application resources to the objectspace.

   All application resources, such as files, services, and programs, must be migrated to the IBM Tivoli Access Manager as protected objects. The protected objects are contained in protected objectspaces. There are various types of protected objects. For every resource, use the object type that is most appropriate for that resource. For information about the various types of protected objectspace and objects, and how to create them, refer to the *IBM Tivoli Access Manager Base Administrator's Guide*. (See “Online resources” on page 422.)

   For your convenience, the two scripts as presented here are included in the downloadable additional material for scenario 1. (See “Additional material” on page 417.)

   The application resource information typically is stored in an application specific ACL database. The DCE security registry can be used for the same purpose. This step has to be performed irrespective of the repository, if you must replace the use of DCE authorization API with aznAPI.

   To demonstrate the migration, consider an application, myapp, using resources file1 and file2. To migrate these resources to the IBM Tivoli Access Manager, create a protected objectspace /myapp of the “Application Container Object (14)” type in the Access Manager, and add file1 and file2 as “file” objects in the objectspace:

   ```
   pdadmin> objectspace create /myapp “Objectspace for my appl.” 14
   ```

```
pdadmin> object create /myapp/file1 "File 1 " 2 ispolicyattachable yes
pdadmin> object create /myapp/file2 "File 1 " 2 ispolicyattachable yes
```

Again, it is convenient to have scripts to do this, if possible.

3. Creating, modifying, and attaching ACLs to the objects added in the objectspace.

   Once the resources are migrated as protected objects in the IBM Tivoli Access Manager, corresponding ACLs must be set on each protected object. The following steps outline what needs to be done:

   a. Create ACLs: Create new access control lists using the **acl create** command. This creates new ACL policies in the ACL database, but not specific ACL entries.

   b. Modify ACLs: Set ACL entries using the **acl modify** command for the users and groups in the specified access control list. The user registry must contain an entry for the specified user or group before you can run this command to add an entry for the user or group to an ACL.

      To set permissions for a user or group, the IBM Tivoli Access Manager defines 17 default permissions. It also provides the capability to define many more additional permissions. You can choose to use the default permissions or create your own as appropriate.

   c. Attach ACLs to objects: Attach the specified ACL to the specified protected object using the **acl attach** command. If the specified protected object already has an ACL attached, this function replaces that ACL with the new one. At most, one ACL can be attached to a given protected object. The same ACL can be attached to multiple protected objects.

   In the example mentioned above, say, file1 has only read permission for a group called mygroup1 and has read, write (modify), and execute permission for mygroup2. To migrate ACL to the IBM Tivoli Access Manager, do the following:

```
pdadmin> acl create ExampleACL
pdadmin> acl modify ExampleACL set group mygroup1 r
pdadmin> acl modify ExampleACL set group mygroup2 rmx
pdadmin> acl attach /myapp/file1 ExampleACL
```

   For detailed information about using the access control policies of the Access Manager, refer *IBM Tivoli Access Manager Base Administrator's Guide*. (See "Online resources" on page 422.)

> **Note:** The examples provided in the previous section used the command line interface to administer the IBM Tivoli Access Manager. This is a suggested way, embedded in script programs, when a large number of objects are to be manipulated. The IBM Tivoli Access Manager also supports a graphical user interface (called the *Web Portal Manager*) that may be more suitable for incidental or help desk operations.

8.3.5 Configuring the Windows Kerberos client

For the revised example in this scenario, a Windows 2000 Professional workstation is configured using an external Kerberos realm for authentication. The Kerberos configuration is provided by the IBM Network Authentication Service and based on the DCE registry data, migrated to LDAP. Thus, the realm's name is the same as the DCE cell name, in this example *realm1.austin.ibm.com*.

To configure a Windows 2000 system into a Kerberos realm, follow the general description provided by Microsoft at the following link:

http://www.microsoft.com/windows2000/techinfo/planning/security/kerbsteps.asp

To perform the actual configuration, the Windows 2000 support command **ksetup** is used. It is part of the Windows 2000 installation CD-ROM. To check whether the Windows System has gained proper Kerberos credentials, the **klist** command, which comes with the Windows 2000 Resource Kit, can be used. To configure Kerberos to accept Windows 2000 as a client, the standard administration command **kadmin** is utilized. The following configuration steps are necessary for this scenario.

Configuration of Kerberos

On AIX, the following administration commands are issued to the IBM Network Authentication Service using **kadmin**:

- After the initial Kerberos configuration, remove references to des3-cbc-sha1 from the default_tkt_encytpes and default_tgs_enctypes lines in the /etc/krb5/krb5.conf file.
- Next, remove both references to des3-cbc-sha1:normal from the /var/krb5/krb5kdc/kdc.conf file.
- Execute **stop.krb5** and then **start.krb5** to refresh the daemons.
- Create the host principal for the Windows 2000 System with this command, using only lower-case letters to specify the Windows system's host name:

```
add_principal host/<windows_hostname>.realm1.austin.ibm.com
```

Configuration of Windows 2000

On the Windows 2000 System, the **ksetup** command is used as follows:

- `ksetup /setdomain realm1.austin.ibm.com`
  This command configures Windows 2000 to use the Kerberos realm realm1.austin.ibm.com as domain. Although the **ksetup** command states that it configures a DNS domain, the realm name given does not have to be an existing DNS or host name.
- `ksetup /addkdc realm1.austin.ibm.com <kdc_hostname>`
  Next, the real, full qualified host name is specified for the node, where the Kerberos server resides. This tells Windows the location of the server, which serves authentication requests for the configured realm.
- `ksetup /mapuser scenario1_princ@realm1.austin.ibm.com local_user`
  The mapuser option of the **ksetup** command links a local Windows 2000 user account with a Kerberos principal. Thus, when you log on to Windows 2000 using principal name scenario1_princ, Windows 2000 uses the specified local user local_user for the local login to the system and scenario1_princ to authenticate against Kerberos.
- `ksetup /setmachpassword <password>`
  This command sets the password that the Windows 2000 system uses to authenticate itself against Kerberos. The password specified in this command has to correspond to the one that was used when creating the host principal in Kerberos. (See “Configuration of Kerberos” on page 149.)

After these commands have been executed successfully, the Windows 2000 System has to be rebooted. After reboot, there is a domain selector in the login dialog. To log on, the domain must be chosen from the list and the user ID (in our scenario1_princ) must be entered with a correct password. When successfully logged on, the **klist** command can be used to confirm that proper credentials have been acquired.

Important: Be careful when replacing login mechanisms on operating systems because errors may lead to unrecoverable situations. The following hints may be useful in order to avoid surprises:

- Preferably, the local user in this configuration should be part of the local machine's administrator group.
- The Kerberos log file /var/krb5/log/krb5kdc.log on the AIX system may have useful information in case of login difficulties or improper credentials.
- The information defined with the **ksetup** command is kept in the Windows registry under the key: `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\Kerberos`. This key can be exported into a registration file for later use.
- Use care when resetting the Windows system to its original domain. The **ksetup** command does not offer a reset option. In this case, the reset may be achieved by entering `ksetup/setdomain` "" and subsequently entering the system into the original workgroup through the **Start** -> **Settings** -> **Control Panel** -> **System** dialog. Do not shut down the system before these steps have been performed successfully, as the system may become unusable otherwise.

Note: This procedure for resetting a Windows system was successful in the lab environment used for writing this book, but it was not found in any product documentation. Therefore, this procedure might not be supported.

8.3.6 Revising the application

Revising the application includes the replacement of the login and authorization code by the respective services provided by the replacement technologies and/or products. See section 8.4, "Revised application discussion" on page 152 for the details concerning these design and programming tasks.

8.3.7 Cleaning up the DCE related information in the IBM Directory

When you are confident that you have migrated all of the users, groups, ACLs on resources, and the application successfully, you can remove DCE-related information in the IBM Directory Server. The appropriate **dcecp** command to remove users and groups supports the option of removing only DCE-specific portions of the registry so that other applications may continue to use the basic principal and group information. This option is specified with the krbdeletetype attribute, so *be sure that the krbdeletetype attribute of the realm entry for DCE in the IBM Directory Server is set to dce(2)* when you delete the DCE principals and

groups using the proper **dcecp** commands. This will remove only the DCE-related attributes from the objects in the IBM DirectoryServer that are not used otherwise. The Kerberos and IBM Tivoli Access Manager information will still be defined and usable.

Examples:

```
dcecp> princ delete <user name>
dcecp> group delete <group name>
```

Finally, when all DCE dependencies are removed, you can also remove the DCE security server from the environment and unconfigure the DCE cell.

8.4 Revised application discussion

In this section, the example introduced in section “DCE application” on page 129, is presented in a revised form with all DCE dependencies replaced. As introduced earlier, the Windows Security Service Provider Interface (SSPI) is used for authentication, and the aznAPI provided by the IBM Tivoli Access Manager is used for authorization. The Windows SSPI is an API that is very similar to the GSS-API, but without providing source code compatibility with the latter. The key services that are used by this API are extracted and explained separately if they have been added or changed compared to the DCE example.

The application logic remains unchanged. The application server returns a string passed by the client in reverse character order.

8.4.1 Configuring and running the revised application

The revised example relies on the same objects as the DCE example (see section 8.2.1, “Configuring and running the DCE application” on page 130), which are now migrated to LDAP and IBM Tivoli Access Manager. In order to comply with IBM Tivoli Access Manager naming conventions, the object scenario1_resource has been renamed to /Scenario1/Resource1 in the IBM Tivoli Access Manager LDAP name space. This represents a resource being protected by the IBM Tivoli Access Manager.

Before making use of the aznAPI, the application has to be configured to communicate with IBM Tivoli Access Manager’s authorization server. (Note that the IBM Tivoli Access Manager documentation sometimes refers t the authorization server as the Policy Director.) This example uses the *remote cache mode* to communicate with authorization servers. This involves less configuration and initialization code at the cost of a possible slight performance decrease. Also, a configuration file is necessary for the example scenario to run; find an example configuration file in “Application configuration file” on page 319. To

configure an application to communicate with IBM Tivoli Access Manager's authorization server, the management command **svrsslcfg** is used. An example for the use of **svrsslcfg** can be found in the example directory that is included in the product installation.

The applications are started simply by entering their executables' names on the command line. Note that most likely changes will be necessary to the hard-coded server hostname in the client example to adapt the program to your environment.

8.4.2 Application client

The revised client example is written for the Windows platform. Section 8.3.5, "Configuring the Windows Kerberos client" on page 149, explains the necessary configuration steps. In the revised application client code, the DCE login code has been eliminated. The program gains the correct credentials from the context provided by the Windows login then initiates a common security context with the application server. Both tasks, acquiring credentials and initializing the security context, are accomplished through calls to the Windows SSPI.

Example 8-10 shows the use of the AcquireCredentialsHandle service in the example program. No principal name is specified, as existing credentials are expected. "Kerberos" is the name of the underlying security package to be used. The SECPKG_CRED_OUTBOUND parameter indicates that the credentials will be used to enable a local credential to prepare an outgoing token. All other in-parameters are omitted. The actual handle for the credential is received in cred_handle. expiry is a pointer to a *timeStamp* structure that receives the time at which the returned credentials expire. This timestamp is not used further in this example, but it could be used in a separate credential refresh thread, which periodically refreshes the credentials before they expire. Such a refresh thread is not implemented in the example.

Example 8-10 SSPI acquire existing credentials

```
maj_stat = AcquireCredentialsHandle(
      NULL,                        // no principal name
      "Kerberos",                  // package name
      SECPKG_CRED_OUTBOUND,
      NULL,                        // no logon id
      NULL,                        // no auth data
      NULL,                        // no get key fn
      NULL,                        // noget key arg
      &cred_handle,
      &expiry);
```

The call to InitializeSecurityContext in Example 8-11 on page 154 closely resembles the GSS-API call in Example 8-2 on page 133. As in the earlier

example it is called in a loop to achieve mutual client/server authentication. cred_handle points to the credentials, obtained in the call to AcquireCredentialsHandle. context_handle is initially empty, but passed in with the value of gss_context in the second call to InitializeSecurityContext. server_name is the principal name that the application server uses. deleg_flag contains the bit flags that indicate the requirements of the context. Those correspond closely to the service options used in Example 8-2 on page 133, although under Windows more options are supported but not used in this example. SECURITY_NATIVE_DREP indicates the data representation, such as byte ordering, on the target. Then, input_desc and output_desc follow; they correspond to rcv_token and snd_token in Example 8-2 on page 133. After two calls to InitializeSecurityContext, gss_context contains a handle to the final security context, shared between the application client and server. The additional output parameters ret_flags and expiry indicate the actual service options that the context offers and the expiration time of the context. Neither is used further in the example.

Example 8-11 SSPI setup of initial security context

```
OM_uint32 deleg_flag = (
   ISC_REQ_MUTUAL_AUTH |
   ISC_REQ_ALLOCATE_MEMORY |
   ISC_REQ_CONFIDENTIALITY |
   ISC_REQ_REPLAY_DETECT);

maj_stat = InitializeSecurityContext(
          &cred_handle,
          context_handle,
          (char *)server_name,
          deleg_flag,
          0,          // reserved
          SECURITY_NATIVE_DREP,
          &input_desc,
          0,          // reserved
          gss_context,
          &output_desc,
          &ret_flags,
          &expiry);
```

Example 8-12 shows the SSPI complement to GSS-API's gss_seal. EncryptMessage takes as input the current security context and as in/out parameter the token to be protected. The two parameters, which are specified with '0', are not used with this version of SSPI.

Example 8-12 SSPI wrapping of messages

```
maj_stat = EncryptMessage(
   security_context,
```

```
0,
&in_buf_desc,
0);
```

8.4.3 Application server

This section explains how the application server example program has to be revised in order to remove DCE dependencies. The necessary changes to the application server's authentication code are actually small. The major change occurs in the authorization part, where the whole DCE authorization module has to be replaced by a module that makes use of the aznAPI. The service provider for the aznAPI is IBM Tivoli Access Manager Policy Director. Finally, as mentioned before, the application logic of the server remains unchanged.

When starting, the server program first gains default credentials by looking up his secret key in the default key table, specifying his principal name in gss_acquire_cred. A separate keytab file could be used here, by setting the environment variable KRB5_KTNAME, before calling gss_acquire_cred. For the principal name used, an entry has to be created in a Kerberos keytab file on the same machine. Again, no Kerberos login is required by the server application as it is purely a context acceptor. As in the DCE example, the server then establishes a common security context with the client. If this is successful, the message exchange between application client and server can take place. The code for all of this remains unchanged. The only change in the authentication code affects the call to gssdce_register_acceptor_identity, described in Example 8-4 on page 134. This call can be removed.

Authorization code

This section explains how the revised server example makes use of the aznAPI. Again, the most relevant service calls are presented in detail. The complete authorization source code can be found in a separate module called azn_authz.c, as opposed to dce_authz.c in the DCE example. (See "Revised application without DCE dependencies" on page 292.)

Example 8-13 provides an excerpt of the function is_client_authorized from the authorization module azn_authz.c. Unlike DCE, the client's authorization information is not passed over the network from the application client, but instead is gained on the server side using aznAPI calls. The principal name of the already authenticated client is used to retrieve its credentials from the underlying authorization system.

Example 8-13 AZN authorization code

```
/* Set mechanism ID to NULL so the aznAPI will determine
 * which registry (LDAP or URAF) that Policy Director is
```

```
    * configured to use. */
   mech = (char *)NULL;
   default_minfo.principal = prin_name;
   buf.length = sizeof(default_minfo);
   buf.value = (unsigned char *)&default_minfo;

   /* create a credential */
   status = azn_creds_create(&creds);
   check_status("azn_creds_create", status);
   if (status != AZN_S_COMPLETE) {
      fprintf(stderr, "Could not create a cred!\n");
      exit(1);
   }

   /* get a credential */
   status = azn_id_get_creds(NULL, mech, &buf, &creds);
   check_status("azn_id_get_creds", status);
   if (status != AZN_S_COMPLETE) {
      fprintf(stderr, "Could not get creds!\n\n");
   }

   /* Perform standard authorization check */
   status = azn_decision_access_allowed(
      creds,
      obj_name,
      operation,
      &permitted);
   check_status("azn_decision_access_allowed", status);

   status = azn_creds_delete(&creds);

   if (permitted == AZN_C_PERMITTED) {
      printf("Permitted.\n\n");
      return TRUE; }
   else {
      printf("Not permitted.\n\n");
      return FALSE;
   }
```

The function is_client_authorized in Example 8-13 on page 155 utilizes two important APIs to achieve authorization:

1. `azn_creds_create and azn_id_get_creds`

   These calls create and extract azn credentials (similar to a DCE PAC) for a given principal.

2. `azn_decision_access_allowed`

This function makes the actual authorization decision by comparing the desired permissions with the client's credential information. In order to achieve this, the application passes in the client credentials, the name of the protected resource, and the desired permissions. A AZN_C_PERMITTED status signals positive permission. Otherwise the request is rejected. Please note that unlike in the DCE example, credential checking and the authorization decision are performed in a single function call.

8.5 Administration considerations and interfaces

This section points out the new considerations concerning administrative tasks and interfaces. The overview comprises the transition environment with DCE security service, the IBM Network Authentication Service, and IBM Tivoli Access Manager all together, and then the final environment without DCE dependencies.

8.5.1 Administration during the migration process

Figure 8-3 on page 158 illustrates the components of the administrative environment during the migration process. The LDAP directory service represents a central repository containing all registry data. The DCE, IBM Network Authentication Service, and IBM Tivoli Access Manager servers all share the data in the LDAP directory.

The LDAP directory as a centralized data store

As a recap, the term *LDAP* refers to a protocol (Lightweight Directory Access Protocol). The term *LDAP directory* denotes a directory that is accessible by the use of LDAP. Besides a few administrative tasks that are done through a Web interface and a server-side plug-in that resides on the LDAP server system, all components access the LDAP directory by means of the LDAP. That means in this sample replacement scenario that the DCE security service, IBM Authentication Service, and IBM Tivoli Access Manager all use LDAP to access the LDAP directory.

In practical environments where security is of primary concern, the LDAP network connection would certainly need to be protected by strong authentication and encryption. (See section 7.4, "Security considerations" on page 113.) LDAP over SSL, simply referred to as *LDAPS*, provides this. See the corresponding product documentation about setting up and using LDAPS.

The DCE registry server uses the LDAP directory to store and manage all data that was kept previously in the local, files-based security registry. The IBM Network Authentication Service uses a subset of the security information kept in

the LDAP directory in order to authenticate users to a Kerberos realm, and additionally for issuing tickets to clients. At last, the IBM Tivoli Access Manager stores and maintains group information, object information, and access control lists in the LDAP directory. The three services share some but not all of their information in the LDAP directory. This raises administration questions as it has to be clear what service is to be used for the various administration tasks.

Figure 8-3 depicts the tree components and their primary administration interfaces that access the shared LDAP directory.

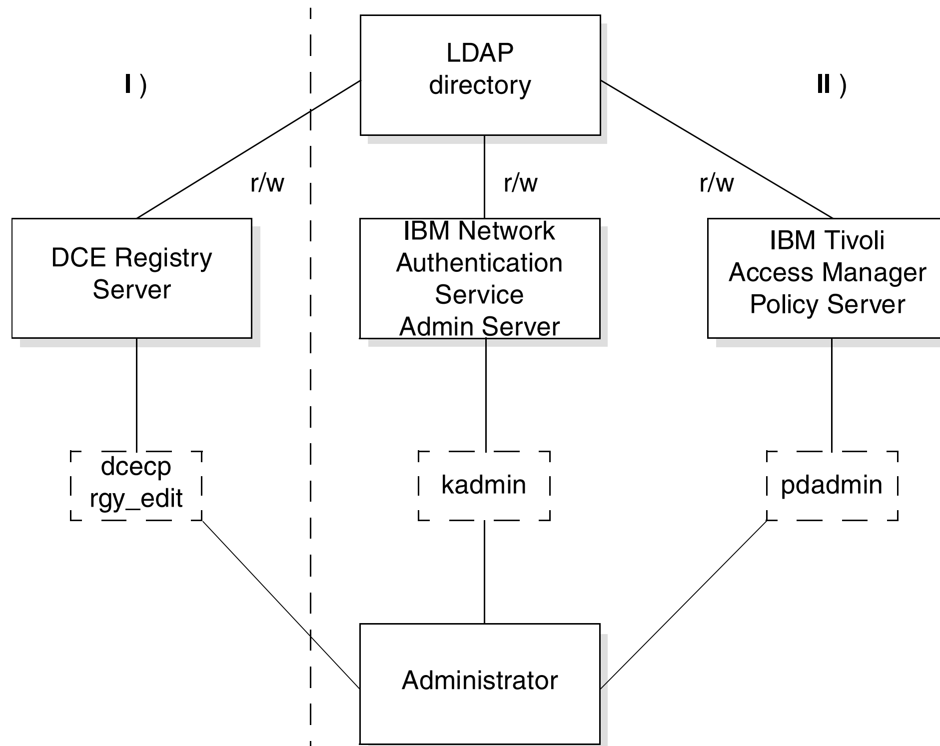


Figure 8-3 Migration environment with DCE dependencies

Sharing data with IBM Tivoli Access Manager

As LDAP information can be modified by every participating component (and, of course, by LDAP-native interfaces as well), some considerations arise concerning administration. As described in detail in Chapter 5, “Using DCE objects with IBM Tivoli Access Manager” on page 85, an administrator has to take care of the consistency of shared objects stored in the LDAP directory, especially when modifications are made by either DCE security service or IBM Tivoli Access Manager.

Each technology shares user objects in the LDAP directory when they refer to same Distinguished Names (DN) of the objects. But only the user name is shared initially while other attributes of the user, such as group membership, passwords, and unique IDs, remain unshared because they are specific to the technologies. If not handled properly, inconsistencies can exist. Another inconsistency over user data can arise because the DCE security service and the IBM Tivoli Access Manager have different ways of handling policies. In DCE, policies are defined in DCE organizations; that is, DCE policies are applied to a user via membership in an organization. The IBM Tivoli Access Manager, on the other hand, creates a default policy and then users can override the policy. Note that it is the task of an administrator to keep consistency between the two technologies.

A newly created DCE user has to be attached to the corresponding IBM Tivoli Access Manager user. On the other hand, DCE security service recognizes IBM Tivoli Access Manager users and the related policy only if they are attached to DCE users.

In the transition environment, it is good practice to change user and group information via DCE first. With the command line interfaces to the DCE security service, including the **dcecp** and **rgy_edit** commands, both user and group information can be managed easily from a single point. Policy data can be administered by means of DCE commands as well. However, keep in mind to update policy and user data for the IBM Tivoli Access Manager user in the LDAP directory every time changes are made. The same procedure applies to group membership information.

Access control lists (ACL) and protected object policies (POP) are maintained by the IBM Tivoli Access Manager differently from DCE ACLs. With the IBM Tivoli Access Manager, administrators protect resources by applying policies to the object representation of these resources in terms of ACLs. Because ACL information is different between DCE and the IBM Tivoli Access Manager, their administration must be done separately for DCE and IBM Tivoli Access Manager.

Sharing data with IBM Network Authentication Service

Sharing objects in the LDAP directory concerning DCE security service and IBM Network Authentication Service turns out to be easier. Following the same LDAP schema layout and realm layout, it is possible for DCE security service and IBM Network Authentication Service to share principals, passwords, and policy information. However, changes that are made to user objects are recognized instantly by both of the technologies.

Replication considerations

Figure 8-4 on page 160 illustrates the replication direction concerning the DCE registry data. The DCE security master, holding the read/write replica of the

registry database, synchronizes with the DCE security replica database that uses the LDAP directory as its database. Changes made to the DCE registry are propagated to the LDAP server database by means of the DCE replication process, therefore are visible for IBM Network Authentication Service and IBM Tivoli Access Manager. On the other hand, if data is added directly to the LDAP database, it is not updated with the master DCE security registry automatically. The administrator has to synchronize both databases by adding the missing data directly to the DCE registry. For the sake of data consistency and simplicity, it is good practice to initiate changes concerning user data via DCE registry.

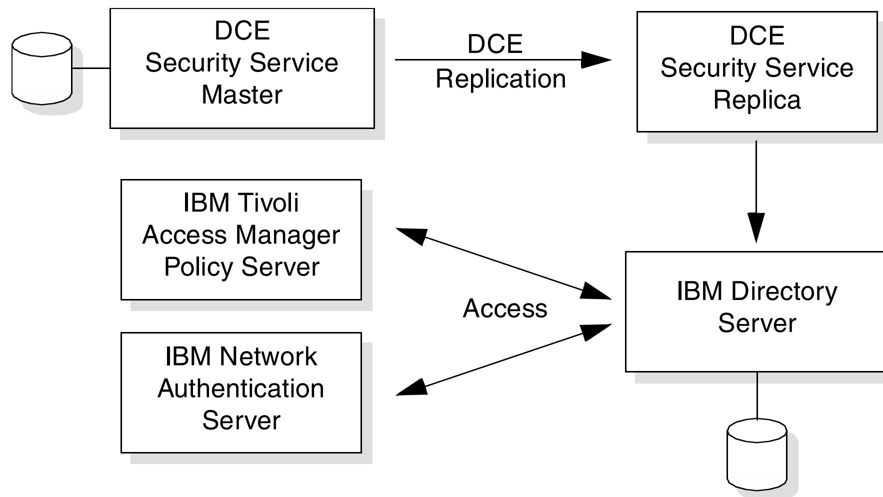


Figure 8-4 Replication and data access in a mixed environment

Best practice approach for common administration tasks

To summarize and better understand the previous statements and the more detailed information given in Chapter 4, “Using DCE data with IBM Network Authentication Service” on page 75 and Chapter 5, “Using DCE objects with IBM Tivoli Access Manager” on page 85, the following sections represent a best practice approach for the most common administrative tasks.

Adding a user

A user is added by performing the following steps:

1. Add a user using DCE commands:

```
dcecp> principal create <user name>
```

2. The previous **dcecp** command creates a new object for the principal in the LDAP directory. In order to attach the DCE user to IBM Tivoli Access Manager, a required objectclass has to be added to the LDAP directory object. IBM Tivoli Access Manager version 3.9 requires the addition of the

inetOrgPerson objectclass to the LDAP directory object; otherwise the import of the DCE user will fail. Newer versions of IBM Tivoli Access Manager may not require this modification. A convenient way to modify the LDAP directory entry is provided by using a temporary file. The following command retrieves the objectclasses of the DCE principal:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind pwd> -b <user DN> \
"krbPrincipalName=*@<realm>" objectclass > user.obj
```

Add the inetOrgPerson objectclass by either editing the user.obj file or executing the following command:

```
echo "objectclass=inetOrgPerson" >> user.obj
```

To modify the LDAP directory information, use the **`ldapmodify`** command:

```
ldapmodify [-h <host>] -D <bind DN> -w <bind pwd> -f user.obj
```

To verify the modification, the **`ldapsearch`** command is issued:

```
ldapsearch [-h <host>] -D <bind DN> -w <bind pwd> -b <user DN> \
"krbPrincipalName=*@<realm>" objectclass
```

3. Import the user using IBM Tivoli Access Manager commands in order to add the attributes that are specific to the IBM Tivoli Access Manager.

   Prior to importing the DCE user to IBM Tivoli Access Manager, you must log on as an IBM Tivoli Access Manager administrator. With IBM Tivoli Access Manager version 3.9, the administrator account is sec_master and cannot be changed. Hence the following command is used to log on:

```
pdadmin> login -a sec_master -p <admin pwd>
```

   Next, the import of the DCE user to IBM Tivoli Access Manager continues:

```
pdadmin> user import <user name> <user DN>
```

   Then the imported IBM Tivoli Access Manager user has to be activated:

```
pdadmin> user modify <user name> account-valid yes
```

   The import of the DCE user can be verified with the **`pdamin`** command:

```
pdadmin> user show <user name>
```

   Note that there is no additional step involved for the IBM Network Authentication Service as it will share the information as added by the DCE commands. However, IBM Network Authentication Service does not recognize the DCE user until an account is created for the DCE user, for instance, via the **`dcecp`** command. Note also that the principal is added to the primary group and the organization at first:

```
dcecp> group add <primary group> -member <user name>
dcecp> organization add <user organization> -member <user name>
dcecp> account create <user name> -group <primary group> -home <user home>\
-organization <user organization> -password <user password>
```

Now the IBM Network Authentication Service can gather information about the DCE user. The following command shows attributes of the user object concerning IBM Network Authentication Service. Note that the **kadmin.local** command can be executed directly on KDC server without Kerberos authentication:

```
kadmin.local -q "getprinc <user name>"
```

Adding a group

Adding a group is simple and can be done following these steps:

1. Add a group using DCE command:

   ```
   dcecp> group create <group name> -inprojlist yes
   ```

2. Import the group using IBM Tivoli Access Manager commands in order to add the attributes that are specific to the IBM Tivoli Access Manager:

   ```
   pdadmin> group import <group name> <group DN>
   ```

   The import of the DCE group can be verified with the **pdamin** command:

   ```
   pdadmin> group show <group name>
   ```

Modifying a group membership

The group membership attribute is shared between DCE and IBM Tivoli Access Manager. If both, DCE user and DCE group, are attached to IBM Tivoli Access Manager, changes concerning group membership via DCE commands are recognized by IBM Tivoli Access Manager.

The following command adds a member to a group in DCE:

```
dcecp> group add <group name> -member <user name>
```

In order to verify that the modification of the group membership is also perceived by IBM Tivoli Access Manager, the **pdamin** command can be used:

```
pdadmin> group show-members <group name>
```

The removal of a user from a group works in a similar way:

```
dcecp> group remove <group name> -member <user name>
```

And again, the **pdamin** command can be used for verification.

Deleting a user or group

Follow these steps to remove a user or a group:

1. Delete the user or group using DCE commands, but be sure the krbdeletetype attribute is set to `dce and krb (3)` so only DCE- and Kerberos-specific attributes will be deleted and the user account remains in the LDAP directory.

   To delete a user in DCE, enter:

   ```
   dcecp> principal delete <user name>
   ```

   To delete a group in DCE, enter:

   ```
   dcecp> group delete <group name>
   ```

2. Delete the user or group using IBM Tivoli Access Manager to remove the user or group and all associated information that is exclusive to it.

   To delete an IBM Tivoli Access Manager user, use the **pdadmin** command:

   ```
   pdadmin> user delete <user name>
   ```

   Delete an IBM Tivoli Access Manager group using the command:

   ```
   pdadmin> group delete <group name>
   ```

> **Note:** No additional step is involved for the IBM Network Authentication Service as the DCE commands also remove the Kerberos information used by the IBM Network Authentication Service. If you do not want DCE to remove the Kerberos information, set the krbdeletetype attribute to `dce (2)`.

Changing a user password

DCE security service and IBM Network Authentication Service can share the password object for users in the LDAP directory. It is good practice to use DCE interfaces to change user passwords. To accomplish this, DCE provides interfaces such as:

- **`dcecp> account modify <account name> -password <new password>`**
- Integrated login in association with the AIX command **passwd**
- `rgy_edit`

Managing authorization data

As mentioned before, the administrator must be aware of the differences between DCE and the IBM Tivoli Access Manager. These differences are likely to be carried on to the applications as well. Thus, an administrator must handle the authorization for each environment separately.

Security

Although this is not normally an ongoing administrative task, it is important for a system administrator to set up and maintain an adequate level of security measures. Specifically:

- DCE, IBM Network Authentication Service, and IBM Tivoli Access Manager should use a strong authentication mechanism, such as SSL with client authentication, to access the IBM Directory Server.
- ACLs should be used and maintained throughout the environments, including DCE, IBM Network Authentication Service, IBM Tivoli Access Manager, and IBM Directory Server, in order to further protect the stored information.
- The system and network environment must be protected physically and logically from unauthorized access.

8.5.2 Administration after the migration process

The right side of Figure 8-3 on page 158 (part II) shows the situation after the migration process is completed. The environment has no more DCE dependencies. The IBM Network Authentication Service and IBM Tivoli Access Manager use the LDAP directory as a shared database for principal information.

As a good practice, users (principals) should be managed only with IBM Tivoli Access Manager. This is, of course, the only way for all other objects that are not known by IBM Network Authentication Service, such as groups and ACLs.

Securing the shared LDAP directory

Administrators must ensure adequate security by maintaining three sets of ACLs:

- In the IBM Network Authentication Service to protect the kadmin server database
- In the IBM Tivoli Access Manager to protect its resources
- In the IBM Directory Server to protect directory objects

Creating a user

Use the following steps to create a new user:

1. Using IBM Tivoli Access Manager interfaces, create an IBM Tivoli Access Manager principal. For example:

   ```
   pdadmin> user create user1
   ```

2. Using LDAP interfaces, search for the DN of the IBM Tivoli Access Manager principal just created (an entry where the principalname attribute equals the name of the IBM Tivoli Access Manager principal).

3. Determine the parent DN of the IBM Tivoli Access Manager principal. For example, if the DN of the IBM Tivoli Access Manager principal is:

   ```
   secAuthority=Default, cn=user1, cn=AM_tree
   ```

   Then the parent DN is:

   ```
   cn=user1, cn=AM_tree
   ```

4. Using the KrbPrincipal auxiliary object class, add the following attribute to the parent DN of the IBM Tivoli Access Manager principal:

   ```
   krbPrincipalName=<princ>@<realm>
   ```

   where <princ> is the Kerberos principal name and <realm> is the Kerberos realm name. For example:

   ```
   krbPrincipalName=user1@realm1
   ```

5. Using the IBM Network Authentication Service interfaces, create a Kerberos principal using the same <princ>@<realm> name that you configured in the krbPrincipalName attribute. For example:

   ```
   kadmin> add_principal user1@realm1
   ```

Deleting a user

Use IBM Tivoli Access Manager interfaces (such as **pdadmin**) to delete the user.

Modifying a user

If you want to modify IBM Network Authentication Service information, use IBM Network Authentication Service interfaces (such as **kadmin**) to make the modification. If you want to modify IBM Tivoli Access Manager information, use the IBM Tivoli Access Manager interfaces (such as **pdadmin**) to make the modification. For example, if you want to change the user password that is used to perform a login to the IBM Network Authentication Service KDC server, make this modification using the IBM Network Authentication Service interfaces.

The two sections that follow give a brief overview of the administration interfaces of the IBM Network Authentication Service and the IBM Tivoli Access Manager.

8.5.3 IBM Network Authentication Service administration interface

IBM Network Authentication Service provides mainly two components:

- The *Key Distribution Center* (KDC) authenticates clients and issues TGTs for requesting service tickets. The KDC daemon is krb5kdc, which listens on the Kerberos port 88. All communication to the LDAP directory resulting from requests by clients is done via the administration server process.
- The *administration server* is used for all kinds of principal administration, which includes adding, deleting, modifying, and viewing principals and

policies. If changes are made, the administration server propagates the changes to the LDAP directory. The kadmind represents the administration server daemon with 749 as the default port. It is required to configure one administration server in a realm.

The interfaces to the administration server are the **kadmin** and the **kdb5_util** commands. Both commands are used to alter information in the IBM Network Authentication Service database. For instance, the **kadmin** command is used to add, delete, and modify principals and administrating keytab files. The **kadmin** command exists as both a remote and a local client (**kadmin.local**). Both **kadmin** and **kadmin.local** provide identical function; the difference is that **kadmin.local** runs on the KDC and does not use Kerberos authentication. As a remote client, the **kadmin** command uses the Kerberos authentication protocol and an RPC mechanism to achieve a secure administration from anywhere on the network. Access to the administration server can be secured by the aid of IBM Network Authentication Service access control. It utilizes an ACL file for mapping of administration user to administrative requests, which is located in the file /var/krb5/krb5kdc/kadm5.acl. If IBM Network Authentication Service access control is configured, the administration server processes a request from a client only if the client is configured with the correct permissions for this request.

Whereas the **kadmin** command is used for direct principal manipulating operations, the **kdb5_util** interface is used for low level database manipulations. In addition, the **kdb5_util** command includes a command to stash a copy of the master database key in a KDC file to enable it to authenticate itself to the kadmind and krb5kdc daemons at startup.

This list includes a brief description of common IBM Network Authentication Service commands:

config.krb5 Configures IBM Network Authentication Service clients and servers.

kadmin, kadmin.local IBM Network Authentication Service database administration program. The **kadmin.local** command runs only on the KDC master without Kerberos authentication. The **kadmin** command comprises a subset of commands.

krb5_util Enables an administrator to perform low-level maintenance procedures on the IBM Network Authentication Service database.

kdestroy Deletes a credentials cache.

kinit Obtains or renews a ticket-granting ticket.

klist Displays the contents of a credentials cache or a keytable.

kpasswd Changes the password for a given principal name.

ksetup Manages IBM Network Authentication Service entries in the LDAP directory. The ksetup program comprises a subset of commands.

ktutil Enables administrators to read, write, or edit entries in a keytab file.

start.krb5 Starts the IBM Network Authentication Service servers (krb5kdc and kadmind).

stop.krb5 Stops the IBM Network Authentication Service servers.

unconfig.krb5 Unconfigures IBM Network Authentication Service clients and servers.

IBM Network Authentication Service can use various levels of authentication to the IBM Directory Server, but SSL with client authentication is the most secure. Be aware that IBM Network Authentication Service stores its ACLs in flat files.

8.5.4 IBM Tivoli Access Manager administration interface

The IBM Tivoli Access Manager provides two administration interfaces:

- The **pdadmin** command line utility. The **pdadmin** command comprises a subset of commands to change user, groups, ACLs, and policies.
- The Web Portal Manager, providing a graphical user interface.

The security administrator may use either the **pdadmin** command or the Web Portal Manager. The **pdadmin** command provides several advanced management functions that are not available through the Web Portal Manager. Also, it is possible to automate certain management functions by writing scripts that use **pdadmin** commands in-line. The utility is installed as part of the IBM Tivoli Access Manager run-time package.

The IBM Tivoli Access Manager servers communicate with the LDAP directory using native LDAP or *LDAP over SSL* (LDAPS). For security reasons, it is recommended to configure IBM Tivoli Access Manager to use SSL. The components of IBM Tivoli Access Manager, namely the authorization server(s) and the policy server, use SSL to communicate with each other.

Note: The information about the administration interfaces of the IBM Network Authentication Service and the IBM Tivoli Access Manager presented in the previous sections is provided here for your convenience. It is not the intention of this book to replace official product documentation or proper education. Also, it is strongly recommended to do a proper systems and security policy planning prior to installing and using these products.

8.6 Discussion and conclusions

The sample scenario in this chapter shows that when applications that make use of DCE security services via DCE login and GSS-API are migrated, the major task usually is migrating the underlying infrastructure. The programming part mainly includes the migration of the client application to use Windows SSPI instead of GSS-API and to replace the DCE authorization code by aznAPI in the application server. Note that the application client code would have required few or no changes for non-Windows clients when using GSS-API. On non-Windows platforms, the GSS-API code can remain unchanged, but the DCE login code would have to be replaced by a Kerberos login. As described earlier, the sample application in this scenario makes use of the login to Kerberos, carried out through a modified Windows login.

An important step in this scenario is the migration of the DCE registry data to the LDAP directory to be used by IBM Network Authentication Service (Kerberos) and IBM Tivoli Access Manager. As demonstrated in section 8.5, “Administration considerations and interfaces” on page 157, this adds some administrative considerations. Making the data available for the new environment is only a first step. Building up a replacement environment that also is fail-safe, scalable, and well-performing includes more planning and configuration. It includes:

- Replication of the LDAP directory
- Replication of the IBM Network Access Manager Authorization Servers including cache tuning

A consideration of the migrated environment is the lack of a single administration interface. Thus, special care has to be taken when modifying security data via the administration interfaces of DCE, IBM Directory Server, IBM Network Authentication Service, or the IBM Tivoli Access Manager.

The new environment gains remarkable advantages by using powerful products as the replacement technologies, including:

- Scalability of LDAP directories

- Additional authentication features of IBM Network Authentication Service, such as its support for Triple DES (3DES) encryption
- Additional authorization features of the IBM Tivoli Access Manager, such as Web-based authorization and graphical user interfaces for administration

Finally, note that the migration path as shown in this scenario provides advantages both for large DCE configurations with large security registries and for smaller environments, because it enables the two environments to run in parallel for an undetermined period of time. This permits a smooth migration without the typical risks that a “big-bang” migration would carry.

9

Scenario 2: Non-secure RPC application

This chapter presents scenario 2, which demonstrates C/C++ replacement strategies for an application that depends directly on the DCE RPC basics, RPC naming, and RPC endpoint services. The strategies are:

- Leave the entire application in C/C++.
- Replace the DCE services with the following services provided by WebSphere: CORBA IDL, CORBA IIOP, and CORBA COSNaming and JNDI.

Before reading this chapter, be sure to read Chapter 7, “Common replacement considerations” on page 111.

© Copyright IBM Corp. 2003. All rights reserved.

9.1 Scenario description

This chapter explains a second scenario application with DCE dependencies and how the DCE dependencies are removed. The chapter contents is:

- The first section discusses the initial application with its DCE dependencies, followed by an explanation of how the DCE dependencies are replaced in this application.
- The initial application with its DCE dependencies is detailed in section 9.2, “DCE application” on page 177.
- The replacement roadmap is explained in section 9.3, “Replacement roadmap” on page 180.
- Section 9.4, “Revised application discussion” on page 181 discusses the revised application.
- A discussion for the management of the new environment is contained in section 9.5, “Administration considerations and interfaces” on page 192.
- Section 9.6, “Discussion and conclusions” on page 192 concludes the chapter.

9.1.1 Initial application with DCE dependencies

The sample DCE application used in this scenario makes use of DCE RPC services and CDS facilities to establish and use a connection from the application client to the application server.

Figure 9-1 on page 173 details the process of establishing a DCE RPC connection using the CDS functionality, as it is used by the sample scenario application.

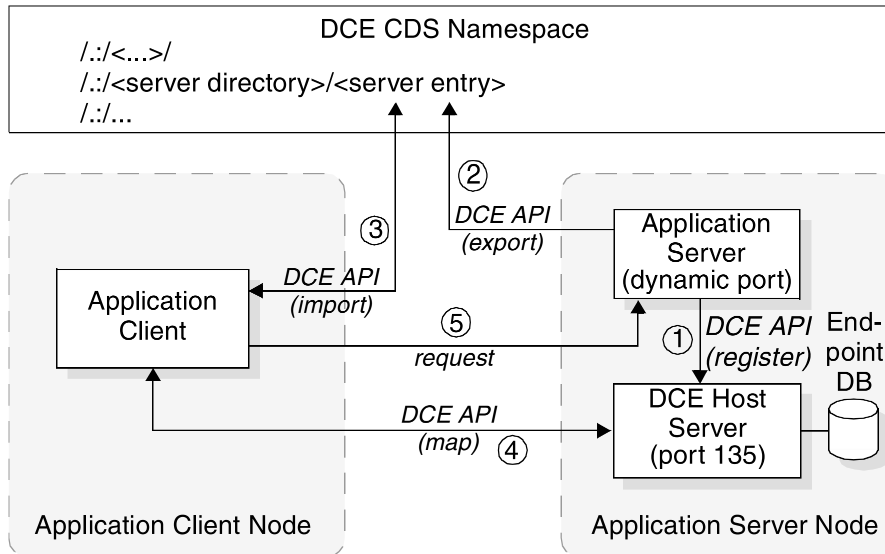


Figure 9-1 DCE RPC connection process using CDS

The different communications that take place in a DCE RPC connection process are described in reference to Figure 9-1:

1. The application server process uses the DCE host server (also called dced) to register its bindings (a structure that contains interface ID, network protocol, network address, port number, and more) with the local endpoint map database. These bindings are initially negotiated with the operating system.
2. The application server exports its bindings to its CDS namespace entry, then begins to listen for client calls at the dynamic ports as registered with the endpoint map. In CDS, only partial bound bindings are stored, including application server node network address and communication protocol.
3. The application client looks up and imports the application server's bindings from the server's CDS name space entry.
4. The application client resolves the partial bindings imported from CDS to full bindings. To do so, the application client contacts the endpoint mapper on the application server node using the information from the partial binding. By passing the interface ID, which the application client shares with the application server, to the endpoint mapper, the endpoint mapper can map the correct endpoints (ports) under which the application server is listening and return it to the application client.
5. Using the full binding obtained in the previous step, the application client can invoke a (remote) procedure call to the correct interface and service offered

by the application server. Note that steps 4 and 5 may be fully transparent to the application client, as DCE RPC run time performs an automatic binding resolution from partially bound to fully bound if required by the programmer.

9.1.2 Revised application without DCE dependencies

For many applications that use DCE RPC services, it seems obvious to consider CORBA for the replacement of the DCE RPC services. This is because of the fact that CORBA, like DCE, knows the concept of *IDL interfaces* as a common definition between application clients and servers. In addition, CORBA, while basically following an object-oriented model, supports applications written in the “C” language, which is the predominant language used for most DCE applications. However, besides these important similarities, CORBA has some differences to DCE, even if only RPC and naming are taken into account.

Some of the most important differences are described below.

RPC issues when migrating DCE RPC applications to CORBA

The features of the IDL of DCE and CORBA show important differences with respect to DCE RPC applications that are to be migrated. While CORBA IDL is designed as a common subset for a broad range of possible supported programming languages, the DCE IDL concentrates on the needs and features of only the C and C++ languages. Hence, DCE IDL offers some important features that CORBA IDL does not, including:

- Full pointer support: DCE IDL enables seamless usage of pointer semantics throughout the network.
- Data streaming, using pipes: DCE IDL offers this feature to transfer a large number of data structures of the same type over the network efficiently.
- Conformant arrays: DCE IDL enables the programmer to specify arrays, the actual size of which can vary from RPC call to call. This enables efficient use of network bandwidth.
- Context handles: DCE IDL enables the programmer to define a context to be transmitted with every RPC call, transparently for the application. The context lets it maintain a state over a series of different RPC calls. RPC run time understands the information in the context and informs the server application if a context becomes invalid.
- Exceptions: In DCE IDL, user exceptions that can be raised by the server implementation code and propagated to the client are defined on the interface level for all services of an interface. In CORBA, user exceptions must be declared as record style data types, containing user-defined fields. Subsequently these exceptions must be specified in every method declaration that could possibly raise them.

IBM WebSphere Application Server's implementation of CORBA 2.3 is no exception to these limitations.

Another important difference between DCE RPC and CORBA lies in the application RPC server itself. A DCE RPC server is, by default, a multi-threaded application, using a configurable number of working threads per exported communication protocol. The thread model used by CORBA, on the other hand, is not standardized but vendor- or platform-specific. Some ORBs are even strictly single-threaded. The ORB provided by WebSphere supports multi-threading. The number of working threads can be configured as a run-time property of the WebSphere CORBA server.

Figure 9-2 gives a high-level overview of how an application client binds to an application server in the CORBA model. The picture follows the process only up to the point where the application client request reaches the server. However, it is important to understand that a CORBA Interoperable Object References (IOR), unlike the DCE binding, carries more details; these include the name of the object adapter, which accepts requests for a particular CORBA object, and the name of the actual CORBA object itself.

Note: The intention of this section is to point out the most important DCE RPC features, which likely must be replaced or changed when migrating a DCE RPC application to CORBA. Therefore, additional features and services that CORBA implementations also may include are not mentioned here.

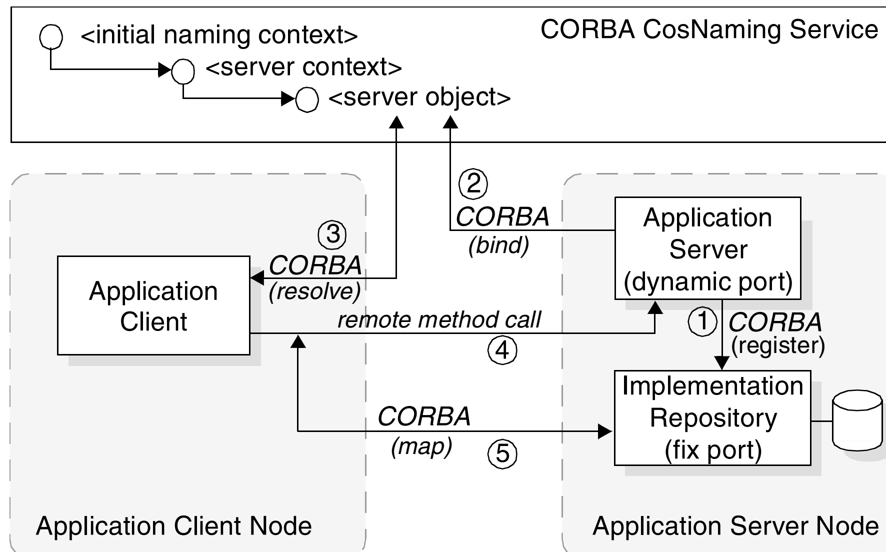


Figure 9-2 CORBA connection process, using CORBA CosNaming service

The different communications that take place in a CORBA connection process are described with reference to the numbers in Figure 9-2 on page 175:

1. If the application server has created a persistent object reference, it registers the object reference with an Implementation Repository, which does not have to reside on the same node as the application server.
2. The application server program binds the object reference with an object name in the CORBA CosNaming service, using its name service context.
3. The application client resolves an object name into an object reference to the application server object using the CORBA CosNaming service.
4. The application client uses the object reference, obtained in the previous step, to invoke a method of the application server object. If the object reference is created as being *transient* by the application server, it points to the actual server object and the request can be processed immediately.
5. In the case when the object reference has been created as being *persistent* by the application server, its network part does not point to the actual server but to the Implementation Repository, which contains the network details (node name, port number) about how to find the actual application server. Using these details, the ORB run time completes the application client's request to the server object. The indirection caused by this extra step is transparent for the application client.

Persistent object references, unlike transient ones, stay valid even in case of an application server termination and restart. The connection process, using an Implementation Repository, is also called *indirect binding*.

IBM WebSphere Application Server Enterprise provides an implementation registration utility, **regimpl**, which must be run in the Implementation Repository to register CORBA servers. For more details, refer to *IBM WebSphere Application Server Enterprise, Version 5, Common Object Request Broker Architecture (CORBA).* See "Related publications" on page 421 for how to download this document.

Naming issues when migrating DCE applications to CORBA

DCE applications that heavily rely on the DCE Cell Directory Service (CDS) may have to undergo some substantial changes when migrating to CORBA. The way the CORBA CosNaming is configured, managed, and programmed against differs from the way it works with DCE CDS. Some of the differences with respect to the DCE applications to be migrated include:

- Groups and profiles: CORBA CosNaming is not designed to support constructs such as groups and profiles to route client accesses to similar network resources in either a random or prioritized way.

- Root contexts: Unlike a DCE CDS cell namespace, CORBA CosNaming allows for more than one root context. (A CORBA CosNaming context can roughly be compared with a directory in CDS.)
- Object UUIDs: CORBA CosNaming does not support DCE UUIDs. In DCE, object UUIDs are used to distinctly identify multiple interchangeable servers by name, for example, or to identify objects inside the server application.
- Management: Some implementations of CORBA CosNaming do not offer management tools. However, the IBM WebSphere Application Server implementation of CORBA offers the **regimpl** tool for management.

WebSphere supports the CORBA CosNaming service, which binds CORBA objects to a public name. Clients are “bootstrapped” according to the CORBA programming model. CORBA-compliant Interoperable Object References (IORs) must be obtained, and server objects must be bound to the CORBA CosNaming service. WebSphere also enables you to manage namespace bindings and name servers through the administrative console.

For a detailed description of IBM WebSphere Application Server name server, refer to chapter 13, “Using naming,” of the *IBM WebSphere Application Server Enterprise, Version 5, Applications*. (See “Related publications” on page 421 to learn how to download this document.)

9.2 DCE application

This section briefly describes the sample DCE dependent program, which makes use of DCE RPC basic, namespace, and endpoint services without using DCE security. The key DCE services that are used by this program are extracted and explained. The complete source code can be found in Appendix B, “Scenario 2: Source code listings” on page 331.

The application logic of the sample application again is simple: The application client sends a string message to the application server in one parameter of the remote procedure call. The application server, after writing the client’s message to standard output, replies to the application client with another string message in another parameter of the same call in reverse character order.

9.2.1 Configuring and running the DCE application

For the example to run, the directory /.:/servers has to exist in the DCE CDS namespace. If it does not already exist, it can be created using the command:

```
cdscp create directory /.:/servers
```

(The issuer of this command has to be logged in to DCE as cell administrator.)

To build the application, the DCE IDL compiler has to process the IDL definition of this example in order to produce a client and a server stub.

Start the application server by entering its executable name on the command line. For the application client, the string message and the number of RPC calls must be added to the command, separated by a blank. For example:

```
$ server_s2
$ client_s2 "Hello World" 5
```

9.2.2 Application client

The client program uses a sequence of calls to the DCE naming service API (rpc_ns_*) to obtain a partial bound bind handle to the server. It exclusively looks for a binding using the UDP protocol. When it finds a binding handle to the server, it immediately invokes an RPC to that server, leaving the endpoint resolution to the DCE RPC run time. The client sample application uses to these key services to make a connection to the server:

- `rpc_ns_binding_import_begin`

  When calling this API, the application client passes in the server's entry name and the interface handle of the interface to be imported. The service returns an import context handle to be used in the subsequent calls.

- `rpc_ns_binding_import_next`

  With the import context handle, created with the previous API, the application client parses the namespace for server bindings, meeting its needs. In this case, a binding handle to the application server is using the protocol UDP. The application client calls this API in a loop until either no more entries are available or a suitable binding is found. To filter out binding handles that use UDP, the client converts the retrieved bindings into string form and performs a string comparison.

- `rpc_ns_binding_import_done`

  After successfully retrieving a binding handle to the server, the application client closes the import context using `rpc_ns_binding_import_done`.

9.2.3 Application server

The application server uses a sequence of calls to the RPC server, RPC naming service, and RPC endpoint API (rpc_server_*, rpc_ns_*, and rpc_ep_*, respectively), using only the UDP protocol. It exports a partial binding to its DCE CDS name service entry /.:/servers/greet and full bindings to the local endpoint map. When this is accomplished, it starts listening for client calls. The server program in this example accepts a maximum of 5 concurrent client connections.

The application server in this sample uses the following key services to register with DCE CDS and RPC in order to receive client calls:

- `rpc_network_is_protseq_valid`

  This API enables the application server to test whether a protocol it wants to use is supported by DCE run time.

- `rpc_server_use_protseq`

  The application server uses this API to create a dynamic endpoint for the requested protocol sequence.

- `rpc_server_register_if`

  Calling this API, the application server registers its interface with RPC run time.

- `rpc_server_inq_bindings`

  Now, the application server can retrieve all binding handles that resulted from the calls it made before. It uses these binding handles in the next two calls to register its binding information with the DCE CDS naming service and the DCE RPC endpoint map.

- `rpc_ep_register`

  This API registers a vector of binding handles, which is passed in, with the local RPC endpoint map.

- `rpc_ns_binding_export`

  This API registers a vector of binding handles, which is passed in, with the DCE CDS naming service.

- `rpc_server_listen`

  When calling this API, the application server is ready to receive client requests. Even though other application server threads might be processing client requests, a call to this function does not return until an rpc_mgmt_stop_server_listening call is issued by another thread or program.

> **Note:** Strictly speaking, the application server in this example is both a server and a client to the DCE CDS. Therefore the server actually has to perform a login to DCE in order to assume a proper identity, which is reflected in the ACL of its name service entry. However, as this example demonstrates non-secure RPC, the code for DCE login is omitted. To run the example, the server would have to inherit proper credentials from its environment (such as by a preceding DCE login on the command line).

9.3 Replacement roadmap

This section describes the migration path for the DCE application to remove the DCE dependencies. In this scenario, the environment (DCE CDS and RPC) is replaced by using the CORBA support provided by IBM WebSphere Application Server Enterprise. The migration roadmap comprises the following steps, which are explained in the sections that follow:

1. Fulfilling the software requirements
2. Installing and configuring the IBM WebSphere Application Server Enterprise
3. Revising the application
4. Removing DCE

9.3.1 Software requirements

Before starting to migrate the application, install and configure the following software:

- WebSphere Application Server Enterprise, Version 5
- Database software (optional). You will need this only if you plan to persist the interface definitions. IBM WebSphere comes with database software.

9.3.2 Installing and configuring WebSphere Application Server

Follow the instructions that come with the product to install IBM WebSphere Application Server Enterprise, being sure to include the following components in the installation:

- CORBA C++ SDK component.
- CORBA C++ SDK samples component.
- You can choose Interface Repository Support if you want to provide persistent storage of interface definitions. If you select this, be sure IBM DB2 is already installed.

9.3.3 Revising the application

In the application code, all of the DCE RPC API calls must be replaced with CORBA API calls. The revised application presented in the next section provides guidelines for revising the DCE application. The entries of application servers in the CDS namespace can no longer be used or migrated. The revised CORBA application server must be registered manually in the CORBA Implementation Repository.

9.3.4 Removing DCE

By removing and replacing the DCE RPC API calls, the DCE dependencies for RPC and naming are also removed. As mentioned before, there is no migration of any DCE data related to RPC and naming, such as endpoint maps to application servers, involved in this scenario.

Unless there are other DCE dependencies, such as for login or authorization, the DCE environment is no longer required to run the revised application.

9.4 Revised application discussion

In this section, the example introduced in section 9.1.1, "Initial application with DCE dependencies" on page 172 is presented in a revised form with all DCE dependencies replaced. The RPC basic, namespace, and endpoint API have been replaced with CORBA naming API provided by IBM WebSphere Application Server Enterprise, Version 5.0. WebSphere supports CORBA 2.3.

In the sections that follow, the key services of CORBA that are used are extracted from the revised example and compared with the DCE example. For a complete code listing of the revised example, refer to Appendix B, "Scenario 2: Source code listings" on page 331.

The application logic remains the same: The application server returns a string passed by the client in reverse character order. In fact, the revised application uses the application logic part of the DCE application and only replaces the RPC portion of it with CORBA.

9.4.1 Building, configuring, and running the revised application

Before starting to configure the revised application, make sure:

- WebSphere Application Server is installed along with the CORBA C++ Software Development Kit (SDK).
- A C++ compiler supported by WebSphere is installed.
- WebSphere Application Server is running.

For details about creating a CORBA application server and client, refer to the document *WebSphere Application Server Enterprise, Version 5, Common Object*

Request Broker Architecture (CORBA). To build, configure, and run the revised application:

1. Create your CORBA application server and client and build them.

   For the revised application presented here, makefiles are provided for building the application. (See Appendix B, "Scenario 2: Source code listings" on page 331.) Use the GreetNt.mak file for Windows and GreetAix.mak file for AIX. You can also choose to build either the application client or the application server or both, for example:

   On Windows:

   ```
   >nmake -f GreetNt.mak GreetClient.exe
   ```

   On AIX:

   ```
   >make -f GreetAix.mak GreetServer.exe
   ```

2. Register the CORBA application server in the WebSphere environment:

   a. Run the following script provided by WebSphere, located under <install root>/bin of the WebSphere Application Server, to set the environment variables:

      On Windows:

      setupCmdLine.bat

      On AIX:

      . setupCmdLine.sh

   b. Enter the following command to store a logical definition for the application server in the Implementation Repository using a server alias. Note that the alias does not have to match the name of the server executable.

      regimpl -A -i <server alias> -p <C++ server>

      For the revised application sample presented here, use the command:

      regimpl -A -i GreetS -p GreetServer

   c. Verify that the Implementation Repository has the server alias defined in it by entering the following command (which gives information about the implementation definition):

      regimpl -L -i myServer

      For example, the output of the **regimpl -L -i GreetS** command for the revised sample application will look like:

      ```
      ============================================================
      Information for implementation definition 1:
         ID:                        2ddd705f-6764-1ec1-e000-0790090304e6
         ALIAS:                     GreetS
         PROGRAM:                   GreetServer
      ```

```
  PROTOCOLS:              TCPIP
  CONFIG:
------------------------------------------------------------
CORB1148I: Exiting the regimpl utility
```

3. Create CORBA application client and CORBA application server properties files. The files contain settings for the values of hostName, bootstrapHostName, bootstrapPort and other run-time properties. The hostName and bootstrapHostName must match your system name, because the CORBA application server must run on the same machine as the WebSphere Application Server. The values set must match the settings used by the application server.

   Refer to the document *WebSphere Application Server Enterprise, Version 5, Common Object Request Broker Architecture (CORBA)* for details about creating properties files.

   For the revised application, modify WSServer.props and WSClient.props to suit your environment. (See Appendix B, “Scenario 2: Source code listings” on page 331.)

4. On the application server machine, set the WASPROPS environment variable to point to the server properties file before running the CORBA application server. In case of the revised sample application, enter the following to set the environment variable and start the application server:

   >export WASPROPS=/scenario2/nondce/WSServer.props (UNIX)

   >GreetServer GreetS

5. On the client machine, set the WASPROPS environment variable to point to the client properties file before running the CORBA application client. In case of the revised application, enter:

   >set WASPROPS=/scenario2/nondce/WSClient.props (Windows)

   >GreetClient IBM 1

6. Use <install root>/bin/**WSStopServer** to stop the application server. The general syntax of the command is:

   WSStopServer <server alias>

The Samples Gallery of IBM WebSphere Application Server provides CORBA C++ samples and instructions about configuring, building and running them. You can use this as a base to develop your applications.

9.4.2 CORBA IDL file

The CORBA IDL file is the public interface to the CORBA application server implementation object class. In the revised application here, it is greet.idl.

Example 9-1 shows the CORBA IDL file. It is analogous to the DCE IDL file that takes an input string from the client and places the output in the “out” string.

Example 9-1 CORBA IDL file

```
interface Greet{
  void greet_CORBA(in string client_greeting, out string server_reply);
};
```

While migrating, you also may choose to change the interface definition. For example, the interface definition could be changed to return a string instead of placing the output in the “out” string:

```
string greet_CORBA(in string client_greeting);
```

For illustration purposes, however, a one-to-one correspondence is maintained in this sample scenario.

9.4.3 Application client

The revised CORBA application client comprises the following:

- Client side usage bindings: These are generated when the IDL files are compiled by the **idlc** compiler. In our example:

  `idlc -suc:hh greet.idl` generates *greet_C.cpp* and *greet.hh*

- The main code for the client program, which in our example is a set of files, is client_s2.c, client_CORBA.cpp, and CWrapper.h.

Each of these components is explained in detail next and compared with the DCE application, excepting the client-side usage bindings as this part is generated by the **idlc** compiler.

Client program

The client of the revised application consists of the following:

- client_CORBA.cpp: This is the C++ CORBA application client that initializes the client environment, locates a servant object hosted by the CORBA application server, and calls methods on the servant object.
- CWrapper.h: This header file consists of declarations for the C wrapper functions of the C++ CORBA application client, so that the C application client, client_s2.c, can invoke the functions defined in the C++ CORBA application clients.
- client_s2.c: This is the modified DCE application client. The application logic part has not been changed.

The following three sections provide more description of each of these files.

client_CORBA.cpp

Example 9-2 shows some of the include statements in the source code of the C+ CORBA application client.

- CosNaming.hh: The header file for the CosNaming functions.
- greet.hh: The client-side usage bindings header file for the application server implementation class. This file is created when the greet.idl file is compiled using the **idlc** compiler.
- CWrapper.h: The header file for the C wrapper functions of the C++ CORBA application client. This is explained in section “CWrapper.h” on page 187.

Example 9-2 Include statements for C++ CORBA application client

```
#include <CosNaming.hh>

/* Local includes follow */
#include "greet.hh"
#include "CWrapper.h"
```

Example 9-3 on page 186 illustrates the initialize function, a C wrapper function that performs the following CORBA C++ operations:

- Initializing the client environment: The initialize function first invokes a function called perform_initialization(argc, argv), which in turn invokes the ::CORBA::ORB_init(argc, argv, "DSOM") function to initialize the Object Request Broker(ORB).
- Getting a pointer to the root naming context: The initialize function then invokes a function called get_naming_context(), which in turn:
  - Invokes the resolve_initial_references("NameService") to find the naming service and receives a pointer to it into ::CORBA::ORB_ptr op.
  - Narrows the object pointer, op, to the appropriate object type by invoking the ::CosNaming::NamingContext::_narrow(objPtr) function and receives a pointer into ::CosNaming::NamingContext_ptr rootNameContext.
  - Performs some checks on rootNameContext and releases the original pointer, op, using the release_resources(::CORBA::ORB_ptr op) function, which in turn invokes the ::CORBA::release(op) function.
- Accessing the servant object: The initialize function finally gets a new ::CosNaming::Name for the servant object that is created by the CORBA application server and looks up the servant object with that name in the name space.

Example 9-3 Initialize function of the CORBA application client

```
int initialize( int argc, char* argv[] ) {
  int rc;
  ::CORBA::Object_ptr objPtr;
  ::CosNaming::NamingContext_var rootNameContext = NULL;

  if ( ( rc = perform_initialization( argc, argv ) ) != 0 )
    return rc;

   cout << "Before getting naming context " << endl;

  // Get the root naming context.
  rootNameContext = get_naming_context();
  if ( ::CORBA::is_nil( rootNameContext ) )
    return -1;

  // Find the Greet_Impl created by the server. Look up the
  // object using the complex name of domain.GreetContext.GreetObject1,
  // which is its full name from the root naming context, as created
  // by GreetServer.

  cout << "Before CosNaming" << endl ;
  try {
    // Create a new ::CosNaming::Name to pass to resolve().
    // Construct it as the full three-part complex name starting at legacyRoot.
    ::CosNaming::Name *greetName = new ::CosNaming::Name;
    greetName->length( 3 );
    (*greetName)[0].id = ::CORBA::string_dup( "legacyRoot" );
    (*greetName)[0].kind = ::CORBA::string_dup( "" );
    (*greetName)[1].id = ::CORBA::string_dup( "GreetContext" );
    (*greetName)[1].kind = ::CORBA::string_dup( "" );
    (*greetName)[2].id = ::CORBA::string_dup( "GreetObject1" );
    (*greetName)[2].kind = ::CORBA::string_dup( "" );
    ::CORBA::Object_ptr objPtr = rootNameContext->resolve( *greetName );
    delete greetName;
    liptr = Greet::_narrow( objPtr );
    cout << "After narrow, liptr = " << liptr << endl;
  }
  catch( ::CosNaming::NamingContext::NotFound e ) {
    cerr << "ERROR: resolve threw NotFound" << endl;
    release_resources( op );
    return 0;
  }
  catch( ::CosNaming::NamingContext::CannotProceed e ) {
    cerr << "ERROR: resolve threw CannotProceed" << endl;
    release_resources( op );
    return 0;
  }
```

```
  catch( ::CosNaming::NamingContext::InvalidName e ) {
    cerr << "ERROR: resolve threw InvalidName" << endl;
    release_resources( op );
    return 0;
  }
  catch( ::CORBA::SystemException e ) {
    cerr << "ERROR: resolve rootNameContext threw SystemException"
      << endl;
    release_resources( op );
    return( 0 );
  }

  return 0;
}
```

Example 9-4 shows the greet_CORBA function, which is also a C wrapper function. It invokes the greet_CORBA method of the servant object. The reply parameter of this function is an indirection to a character pointer, because the server_reply of the function, liptr->greet_CORBA(client_greeting, server_reply) is of type ::CORBA::String_out. It has been implemented this way so that the signature corresponds to the greet_RPC call of the DCE application. However, the signature does not necessarily need to match and an application developer can choose to change it.

Example 9-4 C wrapper for the greet_CORBA function

```
void greet_CORBA( char* client_greeting, char** reply ) {
    char* server_reply;
    liptr->greet_CORBA( client_greeting, server_reply );
    *reply=server_reply;
}
```

There is another C wrapper function, void finalize(), which invokes release_resources(::CORBA::ORB_ptr op), which in turn calls ::CORBA::release(op) to deallocate all of the resources used by the program.

CWrapper.h

Example 9-5 on page 188 shows an excerpt from the CWrapper.h header file. For a C application to use C++ objects or routines, a C wrapper function has to be provided in the C++ application. These wrapper functions will be C functions using C++ objects or routines. For such functions, name-mangling has to be turned off by declaring the prototypes for these functions with C linkage. To achieve this, precede each of the C wrapper function prototypes with extern “C” to instruct the linker to use C linkage and not mangle names.

Note: For a C++ program, name mangling transforms the names of entities (classes or functions) so that the names include information about aspects of the entity's type and fully qualified name. Name mangling generates external names that will not clash. For the compiler to do a C style naming, name mangling must be turned off for C functions.

This header file is included in both the CORBA C++ client and the C application client.

Example 9-5 The header file for the C wrapper functions

```
#ifdef __cplusplus
extern "C"{
#endif
  int initialize( int argc, char* argv[] );
  void finalize();
  void greet_CORBA( char* client_greeting, char** reply );
  void greet_mirror( const char * client_greeting, char * server_reply );
#ifdef __cplusplus
}
#endif
```

client_s2.c

Example 9-6 lists the application client logic. Notice that the application logic part of the DCE application has been preserved and only the following changes have been made:

- The CWrapper.h header file has been included so that the application client can use the C wrapper functions defined in the CORBA C++ client.
- The DCE RPC calls have been replaced by the initialize function.
- The greet_RPC function has been replaced by greet_CORBA.
- The finalize function has been invoked to deallocate resources.

Example 9-6 Application logic

```
#include <stdio.h>
#include "CWrapper.h"

int main ( int ac, char* av[] ) {
   int i, MAX_PASS;
   char  *name_to_greet, *reply;

   if ( ac !=3 ){
     fprintf( stderr, "Usage: %s message passes \n",av[0] );
     exit( 1 );
```

```
    }

    /* A call to the CORBA C++ client to initialize the client,
       get naming context and to access the servant object */
    initialize( ac,av );
    printf( "after initialize \n" );

    name_to_greet = av[1];
    MAX_PASS = atoi(av[2]);
    for ( i=1; i<=MAX_PASS; i++ ){
    /* Access the servant object method.
       In the DCE applciation, the function call was:
       greet_RPC(name_to_greet,reply); */
      greet_CORBA(name_to_greet, &reply);
      printf( "The Greet Server said : %s\n", reply );
      fflush( stdout );
    }

    /* A call to the CORBA C++ client to deallocate the resources */
    finalize();
    return 0;
}
```

9.4.4 Application server

The revised CORBA application server comprises the following:

- Server side usage binding files: The CORBA IDL file is compiled using the **idlc** compiler to produce the usage binding files needed to implement and use the servant object.
- Servant implementation files: These files are also generated by compiling the CORBA IDL file using the **idlc** compiler. They should be modified to add the implementation logic.
- The main code for the server program: The server main source file has the code for the methods that the server implements.

In the revised application presented here, the CORBA IDL file, greet.idl, is compiled using the **idlc** compiler to generate the usage binding files and the servant implementation files.

The command **idlc -e:ih:ic:sc greet.idl** generates the files greet.ih, greet_I.cpp, and greet_S.cpp, where:

- greet.ih and greet_I.cpp are the servant implementing files, and
- greet_S.cpp is the server usage binding file and does not need any modification.

Servant implementation

The modifications made to the servant implementation files, *greet.ih* and *greet_I.cpp* are explained below. The *server_manager.c* file of the DCE application is also part of the servant implementing and is also explained.

greet.ih

This is the skeleton implementation header file for the revised application. This file has been modified to add declarations for constructors and destructors. Example 9-7 shows the constructor and destructor declarations that have been added to this file. The methods declared here have been implemented in the associated greet_I.cpp file.

As the application logic itself is simple, it does not require any other methods or private data members. In a more-complex application, other method declarations and definitions of any private data members for the implementation code in the greet_I.cpp file may be included in the greet.ih file as well.

Example 9-7 Modified greet.ih

```
class Greet_Impl : public virtual ::Greet_Skeleton {

   public:
     ::CORBA::Void  greet_CORBA( const char* client_greeting,
                                 ::CORBA::String_out server_reply );

/* The following declarations for the constructor and destructor have been
   added to this generated file */
   public:
     Greet_Impl();
     virtual ~Greet_Impl();
};
```

greet_I.cpp

The code defines the methods that implement the business logic for the server implementation class, Greet_Impl. The declarations of class variables, constructor and destructor for this class are present in the greet.ih header file, as explained in the previous section. Example 9-8 shows an excerpt from the greet_I.cpp file.

Example 9-8 Excerpt from greet_I.cpp

```
#include "CWrapper.h"

::CORBA::Void  Greet_Impl::greet_CORBA( const char* client_greeting,
                                        ::CORBA::String_out server_reply ) {
  // DEVELOPER_NOTE: Provide method implementation
```

```
    char reply[ STR_SZ ];
    greet_mirror( client_greeting, reply );
    server_reply=::CORBA::string_dup( reply );
}
```

The excerpt in Example 9-8 on page 190 shows the implementation of the greet_CORBA(const char* client_greeting, ::CORBA::String_out server_reply) method as defined in the IDL file, greet.idl. This method invokes a C routine, greet_mirror(client_greeting, reply) which is defined in the server_manager.c file of the DCE application. Hence, the application logic part of the DCE application is preserved. The function prototype for greet_mirror is included in the CWrapper.h header file with an extern "C" for C linkage.

server_manager.c

Example 9-9 shows the revised server_manager.c file. Notice that all of the DCE related declarations and definitions have been removed, retaining only the application logic part.

Example 9-9 Revised server_manager.c

```
#include <stdio.h>
#include "CWrapper.h"

void greet_mirror( const char * client_greeting, char * server_reply ) {
   char tmp[STR_SZ];
   int i,msglen;

   msglen = strlen( client_greeting );
   printf( "The client says: %s\n", client_greeting );
   for ( i=0;i<msglen;i++ ) {
     tmp[i] = client_greeting[msglen-i-1];
   }
   tmp[i] = '\0';
   printf( "This I turn around into %s\n",tmp );
   fflush( stdout );
   strncpy( server_reply,tmp,msglen );
}
```

Server program

In the revised application, the server program consists of just one file, server_CORBA.cpp. The application server performs the following tasks:

1. Validates user input
2. Initializes the server environment
3. Accesses naming contexts
4. Names, creates, and binds a servant object

5. Creates a server shutdown object
6. Goes into a wait loop
7. Services requests

Refer to Appendix B, “Scenario 2: Source code listings” on page 331 for the listing of the server program server_CORBA.cpp. It is not listed here because of its length.

> **Note:** The revised application uses Basic Object Adapter (BOA), as the WebSphere C++ ORB does *not* support Portable Object Adapter (POA).

The server_CORBA.cpp file has been coded as explained in the section “Creating the CORBA server main code” in the *WebSphere Application Server Enterprise, Version 5, Common Object Request Broker Architecture (CORBA).* Refer this document for more details.

9.5 Administration considerations and interfaces

The sample scenario in this chapter only uses communication and naming services; it does not involve any user, group, and authorization services. Administration tasks therefore only include the management of the CORBA naming service, which is done with the implementation registration utility (**regimpl**), provided by the IBM WebSphere Application Server Enterprise.

The **regimpl** utility provides primitive operations on the CORBA naming service, such as adding, updating, deleting, and listing entries. (See section 9.4.1, “Building, configuring, and running the revised application” on page 181.) Refer to the product document *IBM WebSphere Application Server Enterprise, Version 5, Common Object Request Broker Architecture (CORBA)* for reference information related to the **regimpl** utility.

9.6 Discussion and conclusions

This scenario demonstrated the migration of a small, non-secure DCE RPC program to CORBA. Due to the simplicity of the example application, this involved only minor programming and easy configuration tasks.

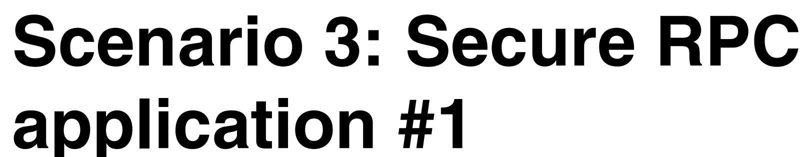
Once a DCE RPC application is successfully ported to CORBA, the wide range of additional CORBA services and facilities provided by some CORBA vendor implementations may well outweigh the effort of the migration. Also, there are mature CORBA products on the market that possess strong failover and scalability features, although different from the ones used by DCE. Another

important feature, which is standardized in CORBA and which many CORBA vendors have added to their product, is a Quality of Service Framework (QoS), which is not in DCE. For example, an RPC time-out mechanism would have to be programmed in DCE and handled by the application using threads and thread cancellation techniques. CORBA QoS enables the application to specify a period of time after which the control shall return from the remote method call.

However, as described in section 9.1.2, “Revised application without DCE dependencies” on page 174, for more elaborated DCE RPC programs, the code migration part can become more complex. Important differences between DCE and CORBA IDL on the one side and DCE CDS and CORBA CosNaming on the other hand lead to this conclusion.

Many features that a DCE RPC application does not need to reflect in its code, such as load-balancing and prioritizing with DCE CDS groups and profiles, must be dealt with in additional code when using CORBA. DCE IDL features that are not present in CORBA IDL must be substituted by lower-level CORBA IDL constructs, which in turn will lead to additional code changes in the application.

10



Scenario 3: Secure RPC application #1

This chapter presents scenario 3, which demonstrates mixed Java and C/C++ replacement strategies for an application that directly depends on the DCE RPC basics, RPC naming, RPC endpoint, RPC security, login, authorization, and PAC services. The strategies are:

- Leave the application client and the business logic of the application server in C/C++.
- Wrap this business logic of the application server in modules coded in Java.
- Replace DCE services with the following services provided by WebSphere: CORBA IDL, CORBA IIOP, CORBA CosNaming, JNDI, LTPA, role-based authorization, CORBA CSIv2, and SSL with client certificates.
- Migrate the DCE registry database to IBM Directory Server, and configure WebSphere to use the migrated data.

Before reading this chapter, make sure that you read Chapter 7, “Common replacement considerations” on page 111.

© Copyright IBM Corp. 2003. All rights reserved.

10.1 Scenario description

The scenario in this chapter demonstrates mixed Java and C replacement strategies for an application that has direct dependencies on secure DCE RPC services as well as the DCE login, authorization, and PAC services. The motivation for using this migration scenario might be for an application with a large amount of existing business logic written in C but with all future development to be in Java. The chapter is organized as follows:

- The first section discusses the initial application with its DCE dependencies and how these dependencies are removed.
- The example application with its DCE dependencies is explained in section 10.2, “DCE application” on page 203.
- Section 10.3, “Replacement roadmap” on page 210 explains the steps necessary to build and run the revised application in the WebSphere environment.
- Section 10.4, “Revised application discussion” on page 230 explains and discusses the revised application.
- Section 10.5, “Administration considerations and interfaces” on page 245 contains some considerations for the management of the new environment.
- Section 10.6, “Discussion and conclusions” on page 246 concludes the chapter.

10.1.1 Initial application with DCE dependencies

The application for this scenario is a simple lookup of employee information that might be found in an on-line corporate directory. This application uses RPC security and authorization to restrict the type of information a directory user can view about an employee.

This scenario uses a highly secure DCE application server and client. The application server ensures that the network security server has validated its credentials before starting. The application client validates the identity of the server before sending it any remote procedure calls (RPCs). When the application client makes RPCs, it annotates the DCE binding handle with the current user’s authentication information. The application server verifies that the RPC was made using packet privacy (encryption) and checks the user’s authorization based on the membership in suitable groups in accordance with the security policy.

The security policy for this sample application is shown in Table 10-1 on page 197.

Table 10-1 Security policy

Function (get)	Unauthenticated	Authenticated Unauthorized	Authenticated Directory Employee	Authenticated Directory Manager
department	deny	allow	allow	allow
grade	deny	deny	allow	allow
salary	deny	deny	deny	allow

Unauthenticated users cannot see any information in the directory. Authenticated but unauthorized users can see only department information. Directory application employees or managers can see more information.

Figure 10-1 depicts the scenario application with the DCE dependencies.

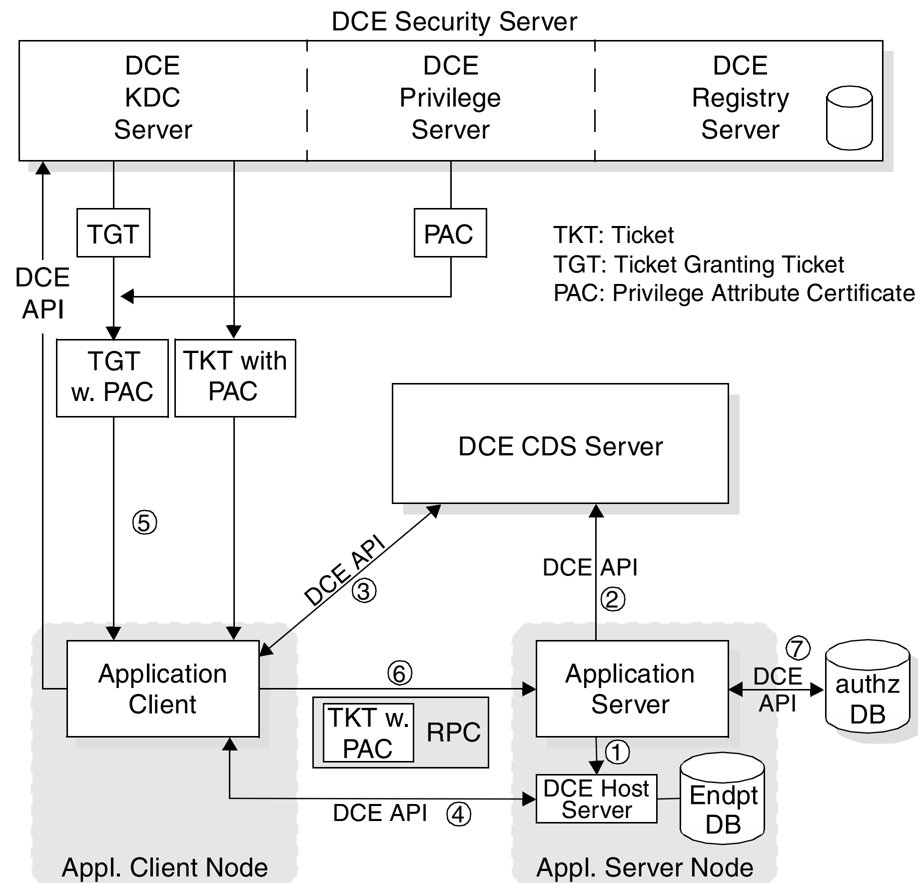


Figure 10-1 Application with DCE dependencies

With reference to Figure 10-1 on page 197, the following simplified steps take place:

1. Upon startup, the application server registers its binding information with the local DCE host server.
2. The application server registers its binding information with the DCE CDS server.
3. The application client queries the DCE CDS server for the partial binding information of the application server.
4. The application client queries the application server node's DCE host server for the full binding information.
5. The application client requests and receives the required tickets from the DCE security server.
6. The application client connects to the application server and forwards the ticket with the PAC information.
7. The application server performs local authorization decisions.

10.1.2 Revised application without DCE dependencies

The replacement strategies to be demonstrated in this scenario are:

- Replacing DCE application client with a CORBA client.
- Replacing DCE application server with a WebSphere Application Server running a stateless session enterprise bean using a JCA connector and JNI to interface with the C business logic. The reason for this choice is that a stateless session bean maps most closely to context-free DCE RPC. DCE RPC using contexts, on the other hand, most likely would be mapped to J2EE stateful session beans.
- Replacing mutual authentication between client and server with the SSL secure exchange identity certificates.
- Replacing DCE authorization with WebSphere J2EE authorization.
- Replacing DCE protection with Secure Sockets Layer (SSL).
- Preserving the business logic in C language.

Figure 10-2 on page 199 illustrates the revised application in the WebSphere Application Server environment.

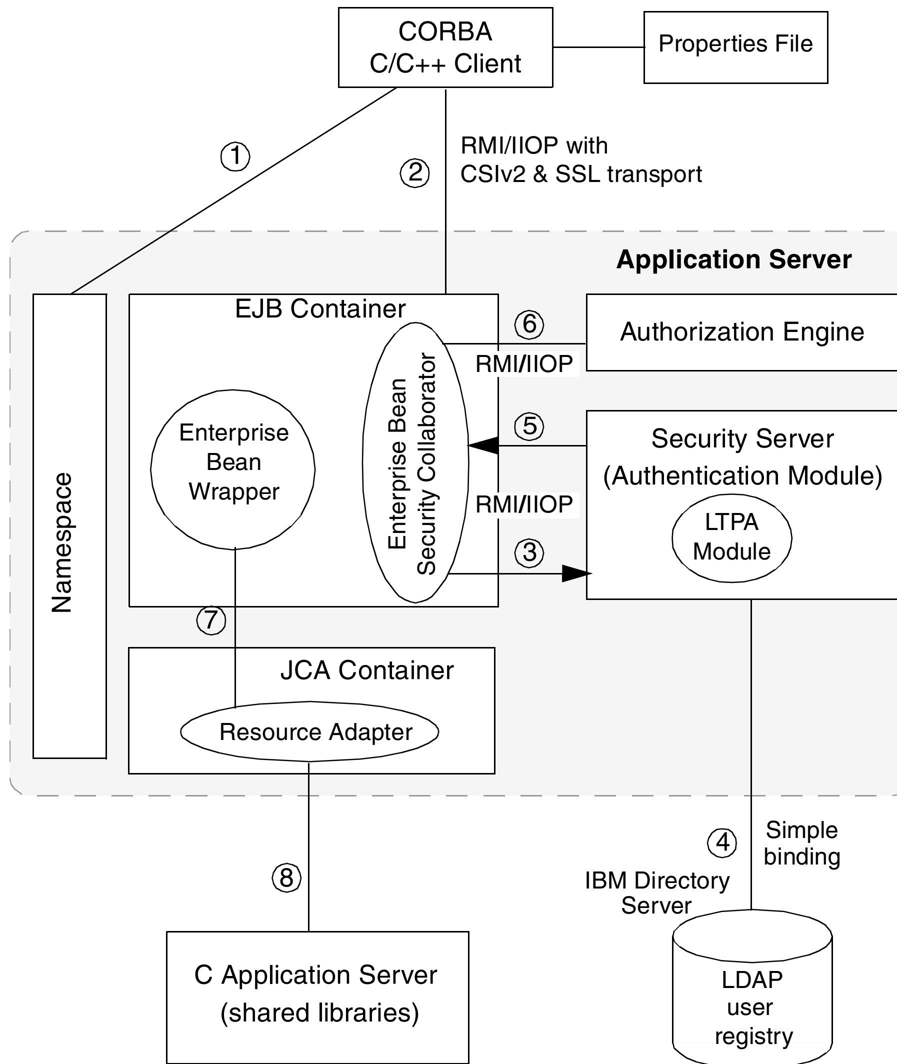


Figure 10-2 The revised application

As shown by the numbers in Figure 10-2, the various steps involved when a CORBA C++ client tries to access the application logic in the C application server via a secured enterprise bean wrapper and a JCA resource adapter are:

1. The CORBA client performs a lookup using CosNaming and receives the Interoperable Object Reference (IOR) of the enterprise bean wrapper.

2. The CORBA client sends authentication information and requests access to the methods of the enterprise bean wrapper. The authentication information is the client certificate sent by means of SSL.
3. The enterprise bean Security Collaborator in the EJB Container sends authentication data to the Security Server and requests authentication.
4. The Security Server performs authentication by looking up the user in the LDAP user registry. In this scenario, this directory has been migrated from the DCE registry.
5. If the user is a valid user, the Security Server returns the credentials for the user.
6. The enterprise bean Security Collaborator performs authorization with the help of the Authorization Engine.
7. Upon successful authorization, the enterprise bean wrapper method calls methods of the resource adapter. For this, the enterprise bean looks up the connection factory of the resource adapter and acquires a JNI connection. It then invokes the respective methods in the connection class.
8. The resource adapter in the JCA container uses the JNI interface to invoke the functions of the native C application server. The return values of the native C functions are returned to the enterprise bean wrapper which in turn returns it to its caller, the CORBA client.

The CORBA client must be configured to access secured applications. Certain properties, such as the security settings of the client ORB, must be provided in a properties file (environment variable WASPROPS must point to this file) or as run-time properties on the command line. Similarly, other properties such as the bootstrap host and port must also be specified.

WebSphere also can be configured to use external authorization systems such as IBM Tivoli Access Manager. For more information, refer to the IBM Redbook *IBM WebSphere V5.0 Security*.

Authentication in the revised application

As mentioned above, authentication of the client is achieved via the information provided by the client's SSL certificate. The subject DN of the client certificate contains the Kerberos name of the user. The identity of the client is verified by a comparison of that subject DN with a Kerberos principal name in the LDAP directory (carried over from the DCE registry migration). This is done by the WebSphere CSIv2 and LTPA authentication mechanisms. The migration of the DCE data to LDAP is used in the previous scenarios and described in 8.3.2, “Migration of DCE security registry to IBM Directory Server” on page 139, or in the document *IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide*.

As a consequence of the revised authentication using client certificates, users must have X.500v3 digital certificates for authentication instead of DCE user IDs and passwords. The matching between the information in the client's certificate and the contents of the LDAP directory can be tailored through the use of customizable filters in order to adapt to individual environments.

Additional information about WebSphere authentication can be found in "WebSphere login and authentication mechanisms" on page 40.

Authorization in the revised application

Authorization is performed in the revised application using standard J2EE authorization mechanisms provided by WebSphere Application Server. (See also "WebSphere authorization mechanisms" on page 41.) Security roles are used to restrict the access to the methods of the Enterprise Bean wrapper. Two roles, *employee* and *manager*, with different authorization options are defined. The manager role can access the methods *getGrade* and *getSalary* but the employee role can access only the *getSalary* method. The revised application does not have any authorization code. During the deployment phase of the enterprise application, a mapping between the roles of the enterprise application and existing groups in IBM Directory Server is done.

Process roadmap overview

The first step to the revised application is to develop enterprise bean wrappers that surround the existing business logic in C. From these, CORBA Interface Definition Language files can be generated that will lead to the production of a CORBA client. Next, a Java class can be created to expose the business logic using the Java Native Interface. Then a set of new C functions can be created to act as the bridge between Java and the old business logic. Finally the enterprise bean can be deployed in WebSphere Application Server.

A roadmap to the overall process is shown in Figure 10-3 on page 202.

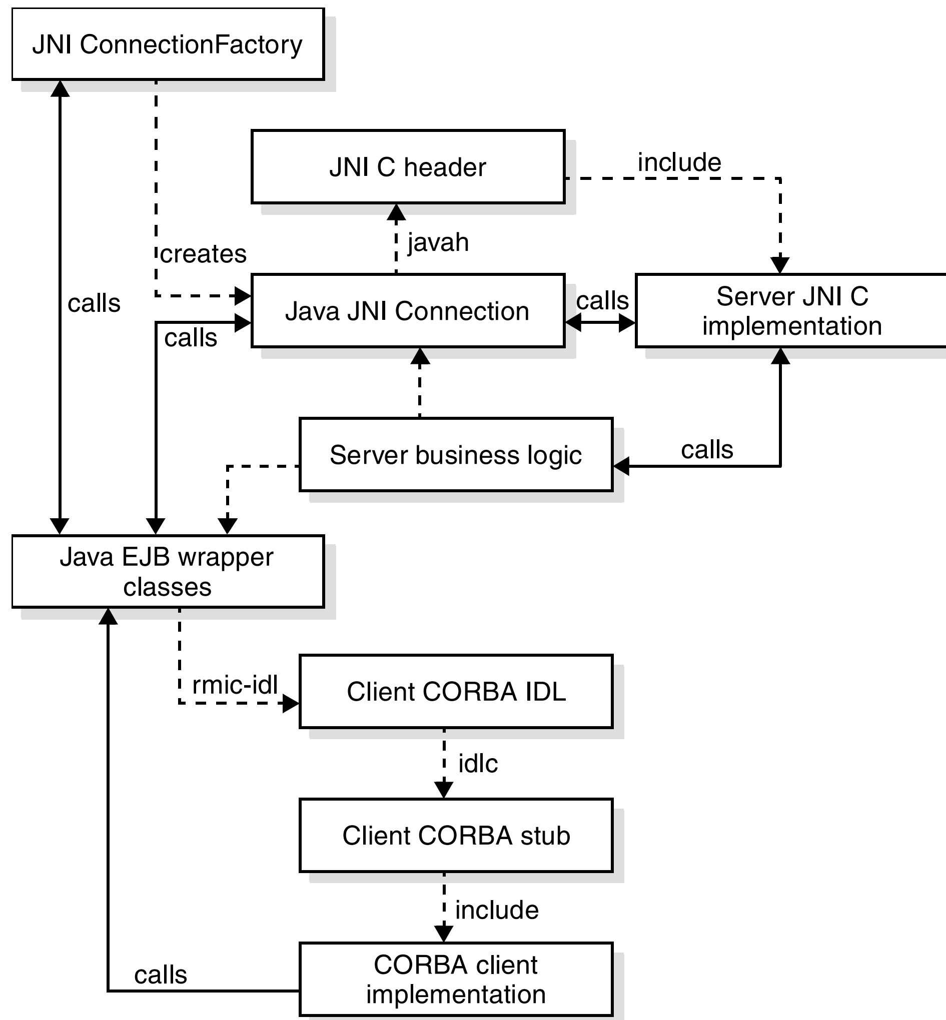


Figure 10-3 Replacement roadmap

The diagram in Figure 10-3 shows the steps required to enable the library of business logic developed in the DCE example to be used in this scenario. From the existing library in the center, there are two manual steps to create an enterprise bean wrapper and a wrapper for using the JNI. The bottom portion of the diagram uses dashed lines from the business logic library to indicate the processes and tools to create CORBA IDL files and supporting C++ classes for building the CORBA client. The upper portion of the diagram also uses dashed lines to show the steps required to produce a new library that implements the JNI for calling the existing business logic. Overlaying this development-based

diagram, shown with solid lines, is an interaction diagram showing the calling sequence from the new CORBA client to the existing business logic.

10.2 DCE application

This section describes the DCE application and how to configure and run it. The dependencies on DCE are explained using excerpts from the program source code (the complete program code listings can be found in Appendix C, “Scenario 3: Source code listings” on page 363).

10.2.1 Configuring and running the DCE application

To set up the DCE example:

- Set up and start the DCE services for directory and security.
- Log on to DCE as the cell administrator.
- Go to the working directory for the directory_server.
- Use **rgy_edit** to enter the following commands, replacing items in angular brackets <…> with information specific for your site:

```
rgy_edit=> domain group
rgy_edit=> add directory_server
rgy_edit=> add directory_employee
rgy_edit=> add directory_manager
rgy_edit=> domain principal
rgy_edit=> add directory_server
rgy_edit=> add john_doe
rgy_edit=> domain account
rgy_edit=> add directory_server -g directory_server -o none \
  -pw <server password> -mp <cell admin pw>
rgy_edit=> add john_doe -g directory_employee -o none \
  -pw <john does's password> -mp <cell admin password>
rgy_edit=> ktadd -p directory_server -pw <server password> \
  -f directory_server_tab
```

- Enable the directory_server principal to create a binding entry in the Cell Directory Service root using **dcecp**:

```
dcecp -c acl modify /.: -add user:directory_server:rwcxid
```

To run the client program you must log on to DCE using the **dce_login** command. The client program takes up to two optional parameters: employee name and *function code*.

For example, this command looks up the department name of the employee john_doe:

```
client -function 0 -employee john_doe
```

There are three function codes that can be used by the example: 0 = get department name, 1 = get employee grade, and 3 = get employee salary. The default function code is 0. There are five employees in the sample directory: peter_morgan, ruth_jones, howard_stein, fran_cooper, and john_doe. The current DCE user is the default employee.

10.2.2 Application interface definition

All DCE applications require an interface definition written in(IDL. This language completely describes the remote function calls to be made from the client to the server. An IDL definition is usually generated manually and stored in a file that ends with the suffix *.idl*.

For this scenario the IDL definition is shown in Example 10-1.

Example 10-1 IDL for sample scenario

```
[
   uuid(e2f98da0-8a19-11d7-8a68-00609437fb07),
   version(1.0),
   pointer_default(ptr)]
interface Directory{
   typedef [string] char *dept_name_t;
   void get_dept(
      [in] handle_t handle,
      [in,string] char *emp_name,
      [out] dept_name_t *dept_name
   );
   void get_grade(
      [in] handle_t handle,
      [in, string] char *emp_name,
      [out]long *grade
   );
   void get_salary(
      [in] handle_t handle,
      [in, string] char *emp_name,
      [out] double *salary
   );
}
```

10.2.3 Application client

The application client is a fairly straightforward DCE program. The code is divided into two sections: initialization and the RPC and presentation logic.

In the initialization code, the application client first looks up the location of the application server from the DCE directory services. It then asks the application server for its DCE network identity and verifies that it is in the directory_server group. The client then creates a binding handle and annotates it with its own DCE credentials. Finally, the application client calls execute_query to perform the requested user function through an RPC. All of this code is shown in Appendix C, “Scenario 3: Source code listings” on page 363.

The RPC and presentation logic is shown in Example 10-2.

Example 10-2 RPC and presentation logic

```
rpc_binding_handle_t binding_handle;
void show_usage(int argc, char *argv[]) {
   printf("Usage is: %s <-function {0|1|2}> <-employee emp_name>",argv[0]);
   exit(1);
}
void execute_query(int argc, char **argv) {
   int i=0;
   int query_type=0;
   enum query_type {DEPT,GRADE,SALARY};
   char *emp_name="";
   long grade;
   double salary;
   char *dept_name="";
   for(i=0;i<argc;i++) {
      if(strcmp(argv[i],"-function")==0) {
         if(i<argc-1) {
            query_type=atoi(argv[i+1]);
            i++;
         }
      }
      if(strcmp(argv[i],"-employee")==0) {
         if(i<argc-1) {
            emp_name=argv[i+1];
            i++;
         }
      }
      if(strcmp(argv[i],"-?")==0) show_usage(argc,argv);
   }
   switch(query_type) {
      case DEPT: {
         get_dept (binding_handle, emp_name, &dept_name);
         printf ("Department for %s is: %s\n",
```

```
            strcmp(emp_name,"")!=0?emp_name:"current user",
            dept_name
         );
         break;
      }
      case GRADE: {
         get_grade (binding_handle, emp_name, &grade);
         if(grade<0) {
            printf (
               "Grade for %s is: unknown\n",
               strcmp(emp_name,"")!=0?emp_name:"current user"
            );
         }
         else {
            printf (
               "Grade for %s is: %i\n",
               strcmp(emp_name,"")!=0?emp_name:"current user",
               grade
            );
         }
         break;
      }
      case SALARY:{
         get_salary(binding_handle, emp_name, &salary);
         if(salary<0) {
            printf (
               "Salary for %s is: unknown\n",
               strcmp(emp_name,"")!=0?emp_name:"current user"
            );
         }
         else {
            printf (
               "Salary for %s is: %f.2\n",
               strcmp(emp_name,"")!=0?emp_name:"current user",
               salary
            );
         }
         break;
      }
      default: {
         printf("Invalid query type %d\n",query_type);
         show_usage(argc,argv);
         break;
      }
   }
}
```

The execute_function function scans the application client argument list for the function or employee parameters and chooses suitable defaults if it does not find them. Then execute_function performs the actual remote procedure call and displays the results.

There are two slightly obscure points in this code. First, the use of the DCE remote binding handle as a global variable, set in the initialization code, is the only dependency in the execute_function function on DCE. By using a global variable, the actual execute_function code remains re-usable. Second, this business logic does not have any code for error handling after the initial contact has been made with the directory server. Production quality code must address this issue.

10.2.4 Application server

The application server code in this scenario is also a traditional DCE application server. The code for the application server has been split into three parts: initialization, RPC handling, and business logic.

For the initialization code, please refer to Appendix C, “Scenario 3: Source code listings” on page 363. The application server first authenticates against the DCE security server to ensure that it is a valid server. Then it binds itself into the DCE directory service under the name directory_server and specifies that clients communicate with it using packet privacy (encryption).

The RPC handling code hides the DCE infrastructure from the business logic and makes that code completely portable. The RPC handling code uses the binding handle from the application client to make an authorization decision in the check_auth function. If it decides that the application client is authorized, it enables the business logic to run; otherwise, it returns some default values to the application client.

Example 10-3 shows an illustrative excerpt from the RPC handling code.

Example 10-3 Sample RPC handling code

```
#include <stdio.h>
#include "Directory.h"
#include "check_status.h"
#include <dce/binding.h>
#include <dce/pgo.h>
#include <dce/secidmap.h>
#include <dce/id_base.h>
#include <string.h>
#include "directory_impl.h"
#define CONTINUE 0
#define UNAUTHENTICATED_USER "unauthenticated user"
```

```
int check_auth(
   rpc_binding_handle_t handle,
   char *type,
   char **client_name
) {
   sec_id_pac_t *pac;
   unsigned_char_t *server_principal_name;
   sec_rgy_name_t client_principal_name;
   unsigned32 protection_level;
   unsigned32 authn_svc;
   unsigned32 authz_svc;
   sec_rgy_handle_t rgy_handle;
   error_status_t status;
   int is_valid=TRUE;
   /*
   * Check the authentication parameters that the
   * client selected for this call.
   */
   rpc_binding_inq_auth_client (
      handle,
      (rpc_authz_handle_t *) &pac,
      &server_principal_name,
      &protection_level,
      &authn_svc,
      &authz_svc,
      &status
   );
   CHECK_STATUS (status, "inq_auth_client failed",CONTINUE);
   /*
   * Make sure that the caller has specified the required
   * level of protection, authentication, and authorization.
   */
   if (! (
      (protection_level == rpc_c_protect_level_pkt_integ) &&
      (authn_svc == rpc_c_authn_dce_secret) &&
      (authz_svc == rpc_c_authz_dce)
   ) ) is_valid=FALSE;
   /*
   * Establish a binding to the registry interface of the
   * Security Server.
   */
   sec_rgy_site_open_query(NULL, &rgy_handle, &status);
   CHECK_STATUS (status, "rgy_site_open failed",CONTINUE);
   /*
   * Convert the UUID in the PAC into a name.
   */
   sec_rgy_pgo_id_to_name (
      rgy_handle,
      sec_rgy_domain_person,
```

```
      &(pac->principal.uuid),
      client_principal_name,
      &status
   );
   CHECK_STATUS (status, "pgo_id_to_name failed",CONTINUE);
   /*
   * Check to see if the client principal is an employee
   */
   if(type!=NULL) {
      is_valid = sec_rgy_pgo_is_member (
         rgy_handle,
         sec_rgy_domain_group,
         type,
         client_principal_name,
         &status
      );
      CHECK_STATUS (status, "is_member failed", CONTINUE);
   }
   /*
   * We are done with the Security registry; free the handle now.
   */
   sec_rgy_site_close(rgy_handle,&status);
   CHECK_STATUS (status, "rgy_site_close failed",CONTINUE);
   *client_name=client_principal_name;
   return is_valid;
}
void IDL_STD_STDCALL get_grade(
   rpc_binding_handle_t handle,
   idl_char *emp_name,
   long *grade
) {
   char *name;
   if(check_auth(handle,"directory_employee",&name)==TRUE) {
      if(emp_name==NULL) emp_name=name;
      *grade=getGradeImpl(emp_name);
      return;
   }
   *grade = -2;
   return;
}
```

The business logic is straightforward. The actual directory is implemented as a global in-memory data structure. The code searches the structure for an employee matching its only input parameter and returns the required value of department, grade, or salary to the application client. In reality, the business logic could be more complicated and could involve retrieving the employee directory as a database query.

Example 10-4 shows an extract of the business logic function.

Example 10-4 Extract from business logic

```
typedef struct {
   char *emp_name;
   char *dept_name;
   long grade_value;
   double salary_value;
} emp_entry_t;

emp_entry_t emp_table [] = {
   {"peter_morgan", "Accounting",1,24000.0},
   {"ruth_jones", "Marketing",2,38000.0},
   {"howard_stein","Finance",2,42000.0},
   {"fran_cooper", "Administration",1,27000.0},
   {"john_doe", "Training",1,18000.0},
   {NULL,NULL,-1,0}
};
DIRECTORY_EXPORT long getGradeImpl(char *name) {
   emp_entry_t *emp_table_p=emp_table;
   int i=0;
   for (i=0;; i++) {
      if(emp_table_p->emp_name==NULL) return -1;
      if (! strcmp (name, emp_table_p->emp_name))
         return emp_table_p->grade_value;
      emp_table_p++;
   }
   return -1;
}
```

By maintaining this code structure for the application server, it is possible to create and deploy the business logic as a shared library on the target platform.

10.3 Replacement roadmap

If the revision of the DCE C code is to work inside a Java-based WebSphere Application Server, it depends on how well the code was structured originally. In our scenario the application server C code was divided into three parts: server initialization, server RPC management and authorization, and business logic. The client was split into two parts: client initialization and business/presentation logic. With this arrangement, it is possible to re-use virtually all of the business logic in both the client and server.

As this type of code usually forms the majority of code in an application, this approach makes the migration highly feasible both in terms of the amount of new

code to be written and in the scope of testing for the revised application. If the application business logic is highly interwoven with DCE infrastructure code, the amount of change required might dictate that a complete rewrite of the application, maybe in Java, is a better approach.

The replacement roadmap includes the following steps, which are described in the sections that follow:

- Meeting the software requirements
- Installing and configuring IBM WebSphere Application Server
- Configuring WebSphere Application Server security
- Configuring the application client
- Developing the application
- Configuring IBM Directory Server (LDAP directory)
- Assembling, deploying, and running the application

10.3.1 Software requirements

The software requirements for running the revised application run time are:

- IBM WebSphere Application Server, Version 5 (any edition can be used except for the CORBA SDK, which is provided with WebSphere Application Server Enterprise only)
- IBM Directory Server 3.2.2
- IBM Java JDK 1.3.1 (included with IBM WebSphere Application Server Enterprise, Version 5)

The software requirements for the revised application development are:

- IBM WebSphere Studio Application Developer 5.01 or other Java development tool
- A C compiler, such as Visual C++ 6.0 for Windows and Visual Age for C++ for AIX

10.3.2 Installing and configuring IBM WebSphere Application Server

The necessary steps for installing and configuring IBM WebSphere Application Server are described in the respective product documentation, and briefly in section 9.3.2, “Installing and configuring WebSphere Application Server” on page 180.

In this scenario, WebSphere Application Server is used on a single system. If multiple WebSphere Application Servers are configured as a cell, install either IBM WebSphere Application Server Network Deployment or IBM WebSphere

Application Server Enterprise. Before installing, check product documentation to find more about WebSphere Application Server Version 5 packaging.

Administration Interface

IBM WebSphere Application Server provides a browser-based graphical user interface for managing all topics related to the Application Server.

When Global Security is disabled, you can reach the administrative console of IBM WebSphere Application Server at:

```
http://<MwWir>:<port>/admin
```

However, when Global Security is enabled, you must use:

```
https://<MwWir>:<secure-port>/admin
```

where:

- `https is` the hypertext transfer protocol over SSL
- `MwWir is` the machine where WebSphere is running
- `secure-port is` the secured port for the administrative console

It is possible to define four different roles concerning administration: *Monitor*, *Configurator*, *Operator*, and *Administrator*. Users who are assigned to the Monitor role are able to view the configuration of the IBM WebSphere Application Server and the current state. Members of the configurator group can alter the IBM WebSphere Application Server configuration. The Operator role has the monitor rights plus the ability to start and stop the IBM WebSphere Application Server. The administrator role comprises both the configurator role and the operator role.

In order to set the administration roles, navigate on the administrative console to **System Administration** -> **Console Users**.

10.3.3 Configuring WebSphere Application Server security

This section describes the necessary steps for setting up the security configuration of the IBM WebSphere Application Server for the current scenario. The components configured in these steps are described in detail in upcoming sections.

1. Prepare SSL key and trust files: These files are needed for the SSL configuration.
2. Configure SSL: This creates an SSL configuration that will be assigned to the CSIv2 communication mechanism.

3. Configure CSIv2: With CSIv2, IBM WebSphere Application Server provides a communication mechanism in order to pass authentication information of the J2EE client to the enterprise application.
4. Configure LDAP as the user registry: As an LDAP directory is used in the current scenario as a user registry, the configuration for accessing the LDAP server is done in this step.
5. Configure LDAP filter rules: The configuration of the LDAP filter rules is needed to match client certificate information to DCE users (principals) in the LDAP directory.
6. Configure LTPA as the authentication mechanism: Specifies the LTPA facility as an authentication mechanism for the IBM WebSphere Application Server.
7. Configure Global Security: To activate the security settings.

Each of the elements must be configured in order to establish a security framework in which the scenario application is secured by the same level as an application is in a DCE environment. Assuming that authorization is done within the enterprise application, authentication, authorization and encryption are covered by the security configuration of the IBM WebSphere Application Server.

Prepare SSL key and trust files

The current scenario uses self-signed certificates created with the **ikeyman** tool, as described in section 7.7.6, “Using self-signed certificates” on page 120. In brief, the following must be done:

1. As a preparation step, copy the default key and trust files provided with WebSphere Application Server to files that are used for the scenario. For example:
   - Copy DummyServerKeyFile.jks to ITSOSrvKeyFile.jks
   - Copy DummyServerTrustFile.jks to ITSOSrvTrustFile.jks
   - Copy DummyKeyRingFile.KDB to ITSOKeyRingFile.KDB

   (On AIX, these default key files are in the /usr/WebSphere/AppServer/etc directory.)
2. Using the **ikeyman** utility and the ITSOKeyRingFile.KDB key file, create a self-signed certificate for the application client. For that certificate, choose a subject Common Name (CN) that matches a DCE principal name. For example, the subject CN of the certificate is *johndoe* for a user whose DCE principal name is *johndoe*. Later, this name will be matched to its

krbPrincipalName attribute in the LDAP directory using LDAP filters (as explained in "Configure LDAP filter rules" on page 218).

The application client key file that is prepared in this step will be used later for the configuration of the application client. (See section 10.3.4, "Configuring the application client" on page 222.)

> **Note:** For step 2, use the **ikeyman** tool that comes with the IBM HTTP Server because this version of **ikeyman** supports the key database file format required for the CORBA client.

3. Extract the self-signed certificate to a file for use in the next step.
4. With the **ikeyman** utility of WebSphere Application Server, add the application client's self-signed certificate to the list of Signer Certificates in the ITSOSrvTrustFile.jks file. This file is used by the Application Server to validate the application client's certificate.

> **Note:** For step 4, use the **ikeyman** utility that comes with WebSphere Application Server because it supports the *jks* file format. On AIX, launch **ikeyman** with **/usr/WebSphere/AppServer/bin/ikeyman.sh**

Because the default key files shipped with WebSphere Application Server already contain the necessary root certificates, no further root certificates must be added to the client's or the server's key files. In a production environment, it is recommended to use certificates from a trusted CA, as shown in scenario 4. (See "Prepare SSL key and trust files" on page 252.)

Configure Secure Socket Layer (SSL)

Navigate on the administrative console to: **Security** -> **SSL**.

The SSL configuration is needed as an authentication mechanism for the client application. For further information and a general discussion about SSL, refer to section 7.7, "SSL implementation hints" on page 116.

Figure 10-4 on page 215 shows the sample settings necessary for the SSL configuration in this scenario. Note that these settings are stored under an alias name that will be referred to in later configuration steps. Multiple aliases (sets of configuration settings) can be created and managed.

General Properties	
Alias	*tivdce3/ITSOSSLSettings
Key File Name	${ROOT}/etc/ITSOSrvKeyFile.jks
Key File Password	*****
Key File Format	JKS
Trust File Name	{ROOT}/etc/ITSOSrvTrustFile.jks
Trust File Password	*****
Trust File Format	JKS
Client Authentication	
Security Level	HIGH

Figure 10-4 SSL settings for IBM WebSphere Application Server

The following list is a description of the items, and how they are used in the current scenario.

- Alias: A descriptive name for the SSL settings, used later as input for the CSIv2 configuration.
- Key File Name: Enter the fully qualified SSL key file name of the key file as prepared in the previous step (“Prepare SSL key and trust files” on page 213).
- Key File Password: The password for the key file specified in the previous entry field. The password was set when the key file was prepared. (The default password for the provided key files is WebAS.)
- Key File Format: Select the database format of the key file. As the default key files were copied and used in this scenario, the file format is JKS.
- Trust File Name: Enter the fully qualified file name of the trust file that was prepared in the previous step (“Prepare SSL key and trust files” on page 213).
- Trust File Password: Specifies the password for the trust file. (The default password for the provided key files is WebAS.)
- Trust File Format: Select the database format of the trust file. As with the key files, the trust files are in the JKS format for this scenario.
- Client Authentication: Specifies whether to request a certificate from the client for authentication purposes when making a connection. This attribute is valid only when used by the Web Container HTTP transport. As IIOP is used for

communication in this scenario, the authentication settings for the client request are made via the CSIv2 Inbound Authentication item, so this item is not selected for the current scenario.

- Security Level: Specifies a selection from a pre-configured set of security levels. Possible values are LOW, MEDIUM, and HIGH. LOW specifies only digital signing ciphers (no encryption). MEDIUM specifies 40-bit ciphers (including digital signing), and HIGH, as used in this scenario, specifies only 128-bit ciphers (including digital signing).

Configure CSIv2

Next, enable the SSL support for the CSIv2 protocol. On the administrative console, navigate to **Security** -> **Authentication Protocol**. Do the following to configure CSIv2:

1. In the CSIv2 Inbound Authentication item, choose **supported** for **Client Certificate Authentication** and select the appropriate SSL alias as defined earlier.
2. In the CSIv2 Inbound Transport item, check the **SSL-Supported** box in the **Transport** field and select the appropriate SSL alias as defined in “Configure Secure Socket Layer (SSL)” on page 214. (If only authentication via client certificate is allowed, set this value to **required** instead of **supported**.)

Configure LDAP as the user registry

Navigate on the administrative console to **Security** -> **User Registries** -> **LDAP**.

This step configures the LDAP user registry that has been migrated from DCE. Figure 10-5 on page 217 shows the settings for the IBM WebSphere Application Server in order to connect to the LDAP server.

General Properties	
Server User ID	* wsadm
Server User Password	* *****
Type	IBM_Directory_Server
Host	* ldap.itsc.austin.ibm.com
Port	389
Base Distinguished Name (DN)	dc=itso,dc=ibm,dc=com
Bind Distinguished Name (DN)	cn=wsldap
Bind Password	*************
Search Timeout	120
Reuse Connection	☑
Ignore Case	☑
SSL Enabled	☐
SSL Configuration	tivdce3/ITSOSSLSettings

Figure 10-5 LDAP settings for IBM WebSphere Application Server

The following list is a description of the items.

- Server User ID: Enter a user ID under which the Application Server runs. Because LDAP directory is used as the user registry in this scenario, the user ID must be a valid user entry in the LDAP directory tree located under the Base Distinguished Name. In the sample scenario, the user wsadm was created with the DCE command **`dcecp user create`**.
- Server User Password: Specifies the password corresponding to the Server User ID as specified in the previous field.
- Type: Select the type of LDAP server to be connected to, which is an IBM Directory Server in our scenario. The type is used to preload default LDAP properties.
- Host: Enter the host name or IP address of the LDAP server. In this scenario, the LDAP server resides on host ldap.itsc.austin.ibm.com.

- Port: Enter the port of the LDAP server. The default port is 389, as used in this scenario.
- Base Distinguished Name: Enter the base DN in the LDAP directory tree used as a starting point for LDAP directory searches. This scenario uses dc=itso,dc=ibm,dc=com.
- Bind Distinguished Name: Enter the DN of the Application Server to use when binding to the LDAP server. This DN must exist (or has to be created) in the LDAP directory with a password (see next option) prior to run the application.
- Bind Password: Enter the password for the Application Server to use when binding to the LDAP server with the DN as entered in the previous field.
- Search Timeout: The scenario uses the default value, which is 120 seconds.
- Reuse Connection: Select if the Application Server should reuse the connection to the LDAP directory server, which is selected (though not required) for this scenario.
- Ignore Case: Specifies that a case-insensitive authorization check will be performed. When using certificates for authentication, as in this scenario, enable this option.
- SSL Enabled: If checked, activates the SSL communication to the LDAP server. In the current scenario, a simple bind to the LDAP server is used for the sake of simplicity, so this box is not checked.
- SSL Configuration: This scenario does not use SSL to connect to the LDAP server (see previous setting *SSL Enabled*), so this entry is ignored.

Configure LDAP filter rules

IBM WebSphere Application Server uses LDAP directory as a repository for users and groups, so appropriate filter rules must be applied in order to resolve users and groups correctly. Filter rules match information in a client certificate to entries in the LDAP directory. Filters are necessary because the representation of the information in a certificate may differ from the information in the directory.

This scenario assumes the name matching as depicted in Figure 10-6 on page 219. Note that johndoe's krbPrincipalName is composed of the DCE principal name and the realm name (johndoe@realm1.austin.ibm.com).

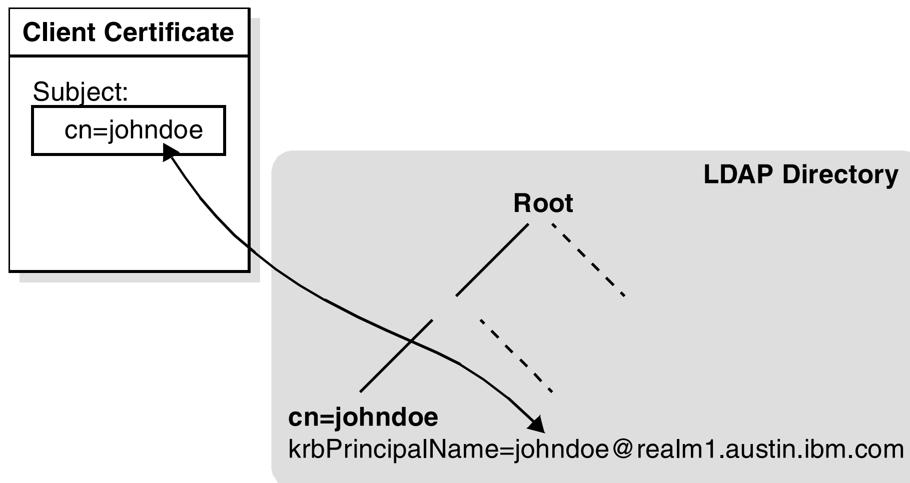


Figure 10-6 Name matching between client certificate and directory entry

As an example, the client certificate's subject CN is *cn=johndoe*, as shown in Figure 10-6. The LDAP filters are configured to match the certificate's subject CN to the krbPrincipalName attribute in the LDAP directory, which in this scenario is *johndoe@realm1.austin.ibm.com*. The krbPrincipalName attribute is set when DCE registry data is migrated to an LDAP directory. The groups residing in LDAP are utilized by the enterprise application to make subsequent authorization decisions based on the group membership of the J2EE client user. Thus, LDAP filters for groups are configured as well.

The configuration of the LDAP filter rules is done via the **Advanced LDAP Settings** item, found in **Security** -> **User Registries** -> **LDAP**. These LDAP filters are used in this scenario:

- User Filter: The LDAP user filter. The value for the current scenario is:

```
(&(krbPrincipalName=%v@realm1.austin.ibm.com)(objectclass=krbPrincipal))
```

- Group Filter: The LDAP group filter. The value for the current scenario is:

```
(&(cn=%v)((objectclass=groupOfNames)(objectclass=groupOfUniqueNames)))
```

- Certificate Map Mode: Specifies whether to match the LDAP entry by the Distinguished Name of the LDAP user or via a specified filter. In the current scenario the value is set to CERTIFICATE_FILTER.
- Certificate Filter: Specifies the LDAP filter, which is used to map the CN in the client certificate to entries in the LDAP registry. The value is set to:

```
krbPrincipalName=${SubjectCN}@realm1.austin.ibm.com
```

Configure LTPA as the authentication mechanism

Navigate on the administrative console to: **Security** -> **Authentication Mechanisms** -> **LTPA**. Enter a password in the **Password** and **Confirm Password** fields. This password will be used to generate LTPA keys when global security is enabled for the first time, as described in the next section, **Configure global security**.

The LTPA represents an authentication mechanism for the security domain of the IBM WebSphere Application Server, whereas CSIv2 (see following section) is utilized as a communication protocol for the J2EE application to the enterprise application.

After a valid authentication of a user, the LTPA mechanism generates security tokens, which can be propagated to subsequent enterprise applications. Equivalent to DCE, the user information can be delegated by means of an LTPA token and application beans within the scope of its authorization role. Acquiring an LTPA token during the authentication process is referred to as the Single SignOn (SSO) feature of IBM WebSphere Application Server. In the current scenario, the LTPA mechanism is used for communication with the user registry located in the LDAP directory.

LTPA uses three keys:

- A public/private key pair for signing LTPA tokens
- A shared key for encrypting the LTPA token

When global security is turned on for the first time with LTPA as the authentication mechanism, LTPA keys are generated automatically. The keys are protected with the password entered in the LTPA configuration panel. This panel (not shown here) is simple, because only the password for protecting the keys has to be specified.

Configure global security

Navigate on the administrative console to: **Security** -> **Global Security**.

Figure 10-7 on page 221 shows the global security settings of the administrative console as used for the current scenario. Beside the main activation of the security configuration, items such as user registry, the active authentication mechanism, and communication protocol are set in this mask. Note that the Java 2 Security is not needed, as global security policies are not utilized in the current scenario.

General Properties	
Enabled	[x]
Enforce Java 2 Security	[ ]
Use Domain Qualified User IDs	[ ]
Cache Timeout	* 600
Issue Permission Warning	[ ]
Active Protocol	CSI and SAS
Active Authentication Mechanism	* LTPA (Light weight Third Party Authentication)
Active User Registry	LDAP

Figure 10-7 Global Security settings for IBM WebSphere Application Server

The following list provides a description of the items.

- Enabled: Must be checked to enable WebSphere global security.
- Enforce Java 2 Security: Used to activate or disable Java 2 security permission checking. The application in the current scenario does not use security policies for simplicity reasons, so this box is not checked.
- Use Domain Qualified User IDs: Not used, and thus, not checked for the current scenario.
- Cache Timeout: The scenario uses the default value.
- Issue Permission Warning: This check box is used in conjunction with Java 2 security and is not used (and thus not checked) in the current scenario.
- Active Protocol: This setting is for backward compatibility with previous versions of IBM WebSphere; the selection does not affect the current scenario.
- Active Authentication Mechanism: Specifies the active authentication mechanism, which is LTPA for the current scenario.
- Active User Registry: Specifies the active user registry, which is LDAP for the current scenario.

After applying and saving the settings, the IBM WebSphere Application Server must be restarted to activate the security settings. Furthermore, the login process to the administrative console requires the name and the password of the console users.

10.3.4 Configuring the application client

The CORBA C++ client is configured after the enterprise application and the client is built. The configuration is explained here for the sake of continuity. Refer to section 10.4.2, “CORBA IDL file” on page 231 and section 10.4.3, “Application client” on page 231, for instructions for building the client.

Complete the following configuration steps before running the client:

- Ensure that you have access to the client key file that was prepared in “Prepare SSL key and trust files” on page 213. The client key file contains the self-signed certificate that is used for client authentication. In this scenario, the file name is:

  `/usr/WebSphere/AppServer/etc/ITSOKeyRingFile.KDB`

- Create a client properties (WASPROPS) file. A CORBA client must have several initialization properties pre-defined. These are used during the ORB init method. These properties may be specified on the CORBA client command line, included in the parameters to the ORB init method, or contained inside a properties file. The name of this special file is specified in the WASPROPS environment variable. The WASPROPS file for the CORBA client in this scenario, `WSEJBClient.props`, is shown in Example 10-5.

Example 10-5 WSEJBClient.props file for the CORBA client

```
com.ibm.CORBA.bootstrapPort=2809
com.ibm.CORBA.translationEnabled=1
com.ibm.CORBA.nativeWCharCodeset=UCS2
com.ibm.websphere.serverName=server1
com.ibm.websphere.EJBName=ejb/com/ibm/redbook/scenario3/EJB/DirectoryHome
com.ibm.CORBA.securityEnabled=yes
com.ibm.ssl.keyFile=/usr/WebSphere/AppServer/etc/ITSOKeyRingFile.KDB
com.ibm.ssl.keyPassword=WebAS
com.ibm.CSI.performTLClientAuthenticationSupported=yes
com.ibm.CSI.performTransportAssocSSLTLSSupported=yes
com.ibm.CORBA.initServices=security
com.ibm.CORBA.dllName=libwassccl.so
com.ibm.CORBA.securityTraceLevel=1
com.ibm.CSI.performMessageIntegritySupported=yes
```

- Set the WASPROPS environment variable to the file created above before running the application client.

After the client configuration has been completed as described, the application client can be run. (See section 10.3.9, "Running the application client" on page 229.)

10.3.5 Developing the application

Application development in J2EE differs from developing DCE applications. Whereas DCE (and CORBA) eased network and security programming by providing IDL and high-level programming APIs, J2EE goes one step farther: It almost completely frees the application developer from programming on the middleware level. The Application Server's middleware API is not accessible for the programmer, but only used by the Application Server itself and the container in which the J2EE application is deployed. Explicit security programming can be avoided mostly by using declarative security; that is, the container handles security as configured during application assembly and deployment.

10.3.6 Configuring IBM Directory Server

As mentioned earlier, the revised application uses the migrated DCE user registry data in the IBM Directory Server. For instructions for the migration, refer to section 8.3.2, "Migration of DCE security registry to IBM Directory Server" on page 139.

If you plan to use BasicAuth as the authentication type for application users, the passwords of all of the migrated users must be set manually (or through a script) in the IBM Directory Server, as WebSphere will not be able to recognize the DCE (Kerberos) passwords. However, this scenario uses client certificates, and thus, the migrated data can be used without any modification. The LDAP filter rules as discussed before provide the necessary means of mapping certificate information to corresponding information in the LDAP directory.

10.3.7 Assembling the scenario application

In the J2EE development process, the responsibility of the application programmer ends with the delivery of a *jar* file that contains the class files of the enterprise beans developed for an application. If a native Java application client has also been developed, then there would be a jar file for that as well.

These jar files are then packaged using an application *assembly tool* that is usually included in the distribution of the Application Server product, which is the case with IBM WebSphere Application Server.

The application assembly tool enables the application assembler to package applications according to the J2EE specification. It lets you create enterprise

bean modules, Web modules, resource adapter archives, and application client modules from the jar or class files, along with the configuration input. These then can be assembled into an enterprise application archive.

The output of the assembly process is a *war* (web archive) or *ear* (enterprise archive) file, which can be deployed using the Application Server's management interface. The war and ear files (without the deployment code) can be deployed on any J2EE Application Server.

General steps

The the most important tasks performed by the Application Assembly Tool (AAT) provided by WebSphere are:

- Generating standard J2EE deployment descriptors, IBM deployment descriptor extensions, and binding definitions for each of the modules. Bindings also can be defined during deployment time.
- Configuring enterprise bean modules with session or entity beans. And, in case of entity beans, declaring container-managed relationships.
- Configuring Web modules with servlets and JSPs. And, in case of servlets, declaring servlet mappings.
- Packaging a client application.
- Declaring environment variables.
- Declaring enterprise bean references and creating resource references.
- Setting enterprise bean transactional attributes.
- Defining security roles and creating a placeholder for the corresponding roles with existing users and groups in the Application Server's security registry.
- Assigning roles to enterprise bean methods in order to control access.
- Assigning bean or Method Level Run-As mappings to enable enterprise beans to call downstream enterprise beans with the same security identity as the original caller ID or with another specified ID(delegation).

After the application assembler has entered all necessary information, the assembly tool can create the XML deployment descriptors for the enterprise beans of the application client and the application itself. These descriptors, together with the extended jar files, are subsequently packed into the ear file, generating the output.

The AAT can generate deployment code as well, such as stubs, skeletons, and helper class files for the WebSphere Application Server. They enable application clients and enterprise beans to communicate over the network in the first place. These generated files are added to the jar file, which contains the enterprise bean class files.

Assembling the scenario application

The following steps assemble the scenario application, called Redbook_Sample. On a system that has WebSphere Application Server installed and that has access to the jar files for the J2EE client and Mirror enterprise beans, do the following to assemble the application:

1. On Windows, start the WebSphere Application Server Application Assembly Tool by selecting **Start** -> **Programs** -> **IBM WebSphere** -> **Application Server v5.0** -> **Application Assembly Tool**.

   On Unix, start the Application Assembly Tool using the shell script:

   ```
   <install root of WebSphere>/AppServer/bin/assembly.sh
   ```

2. Select **Application** from the choice of subjects that can be created.
3. In the panel for the new enterprise application, enter **Redbook_Sample** for the Display name, and click **Apply**.
4. On the left part of the Assembly Tools window, select **EJB Modules** and right-click to select **New**.
5. Enter the common configuration information for the enterprise bean:
   a. Replace the default jar file name with `Redbook_Sample_EJB.jar`.
   b. Enter the Display Name `Redbook_Sample_EJB`.
6. Add the enterprise bean directory to the created EJB Module and configure it:
   a. Under EJB Modules, expand the **Redbook_Sample_EJB** entry, select **Session Beans**, and right-click to select **New**.
   b. In the General tab of the New Session Bean window, enter `Directory` as the EJB Name and subsequently browse and enter EJB class, EJB home, and remote interface, through accessing the EJB.jar file for the DirectorySession provided by the application development process. Enter:
      - `com.ibm.redbook.scenario3.EJB.DirectorySessionBeanImpl.class` for EJB Class
      - `com.ibm.redbook.scenario3.EJB DirectorySessionHome.class` for Home
      - `com.ibm.redbook.scenario3.EJB DirectorySessionBean.class` for Interface
   c. Select the Advanced tab and ensure that **stateless** is selected for the Session Type.
   d. Select the Bindings tab and enter the JNDI name for Directory, which in this case is `com/ibm/redbook/scenario3/EJB/DirectoryHome`.
   e. Select **OK** in the General tab.

7. Return to the application level on the left side of the main window, select **Security roles**, and right-click to select **New**.
8. Create two new roles, *manager* and *employee*. In the Bindings tab of the New Security Role window, bind the manager role with registry group `cn=DirectoryManagers,cn=group,krbRealmName-v2=realm1.austin.ibm.com, dc=itso,dc=ibm,dc=com`
9. Bind the employee role with registry group `cn=DirectoryEmployees,cn=group,krbRealmName-v2=realm1.austin.ibm.com ,dc=itso,dc=ibm,dc=com`

   In the scenario example, these groups have been created by DCE and migrated to the LDAP directory.

   Note that you must use the full Distinguished Name (DN) of the groups with the assembly tool. When this is done, the role manager and employee are known globally for the Directory enterprise application and are ready to be used for further configuration.
10. Specify for the EJB Module Redbook_Sample_EJB which roles to use locally. To do so, under EJB Modules, expand the **Redbook_Sample_EJB** entry, and right-click **Security roles** to select **New** twice to create the roles *manager* and *employee* here as well.
11. Now the configuration is ready to annotate enterprise bean methods with role-based permissions. Therefore, on level **Redbook_Sample** -> **Redbook_Sample_EJB** -> **Session Beans**, select **Method Permissions** and right-click to create new permissions.
12. In the New Method Permission window, enter a method permission name of `manager`. Then click **Add** for Methods and select the method to be annotated. In our scenario, this is the *getGrade* and *getSalary* methods of the Directory enterprise bean. These methods contain the business logic of our example for manager use. Add `getGrade` and `getSalary` and click **OK**. Next, click **Add** for Roles and select the manager roles that shall have access to the methods. Click **OK**.
13. Again in the New Method Permission window, enter a method permission name of `employee`. Then click **Add** for Methods and select the method to be annotated. In our scenario, this is the *getGrade* method of the Directory enterprise bean. This method contains the business logic of our example for employee use. Add `getGrade` and click **OK**. Next, click **Add** for Roles and select the employee roles that should have access to the methods. Click **OK**.
14. Create a resource reference to link the EJB with the resource adapter: Go to **RedbookSample** -> **Redbook_Sample_EJB** -> **SessionBeans** -> **Directory** -> **Resource References** and right-click to create a new reference. Name the reference `redbook_jni_connection` and type `javax.resource.cci.ConnectionFactory`. Choose **Application** for

Authentication. Click the Bindings tab and enter `redbook_jni_connection`. Then click **OK** to save the reference.

15. Create a resource adapter: On the left part of the Assembly Tools window, select **Resource Adapter** and right-click to select **New**.
16. Enter the common configuration information for the resource adapter:

    Click the General tab. Enter `Redbook Sample Connector` for the Display name, `JNI Connection` for EIS type, `IBM` for Vendor name, `1.0` for Version, and `1.0` for Specification.
17. Click the Advanced tab and browse for the Connector.jar provided by the application development process. Enter:
    - `com.ibm.redbook.scenario3.RA.JNIManagedConnectionFactory.class` for ManagedConnectionFactory
    - `javax.resource.cci.ConnectionFactory` for Connection Factory interface
    - `com.ibm.redbook.scenario3.RA.JNIConnectionFactory` for Connection Factory implementation
    - `javax.resource.cci.Connection` for Connection interface
    - `com.ibm.redbook.scenario3.RA.JNIConnection` for Connection implementation

    Click **Apply** to save these entries.
18. From the left panel, select **Files** and right-click to select **Add**. Browse to the Connector.jar file and select all files to add. Click **Add** and **OK**.
19. Go to **File** -> **Save As** and save the application archive as `Redbook_Sample.ear`.

10.3.8 Deploying and starting the application

After the enterprise application has been successfully assembled and the ear file is produced, the application can be deployed using the Application Server's Management interface. (In the WebSphere Application Server product, this is the Administrative Console.) Application Deployment or the role of an application deployer in J2EE is a subject that has no well-defined counterpart in the DCE application development process. Whereas J2EE application deployment is all about tool-based configuration, deploying a DCE application would be a combination of DCE configuration, script programming, and system management tasks. Due to the specialities of the Java programming language (reflection, dynamic class loading), in the J2EE deployment phase there are configuration options, which correspond to the application programming phase in a DCE environment. For example, in J2EE, it is possible to modify access right to enterprise bean methods even in the deployment phase.

General tasks

These important tasks are carried out in the deployment phase:

- Generating default bindings (containing EJB JNDI names, EJB references, resource reference bindings, connection factory bindings, and data source bindings) or specifying a bindings file.
- Setting JNDI name and type of resource authorization for default data sources (DB access) used by entity beans with container managed persistence.
- Enabling enterprise application options such as using JSP precompilation and class reloading.
- Mapping user registry objects such as users and groups to enterprise application security roles. This can also be done during the assembly phase.
- Managing mappings of deployed modules or applications to a server or a server cluster.
- Most important, generating deployment code and installing an Enterprise Application once all the settings are specified.

After all necessary configuration information is entered, the deployment tool saves the configuration and installs the enterprise application accordingly on the participating J2EE Application Servers.

Deploying the sample application

These steps deploy and start the scenario application, the sample enterprise used in this scenario. As a prerequisite, be sure that the Enterprise Application Archive file Redbook_Sample.ear is accessible, then open the WebSphere Administrative Console. Perform the following steps to deploy the application Redbook_Sample:

1. The left side of the Administrative Console displays a list of subjects that can be configured for each Application Server. In our example there is only one instance of an Application Server. Expand the ‘+’ on **Applications** and click **Install New Application**.
2. On the right side of the console window, enter the local or remote path to Redbook_Samplw.ear and click **Next**.
3. Check **Generate general bindings** in the current page and click **Next**.
4. On this panel, check **Deploy EJBs**. Click **Next**.
5. No further deployment options are necessary in this panel. Click **Next**.
6. This panel should already contain the JNDI Name for Directory, com/ibm/redbook/scenario3/EJB/DirectoryHome. Leave it unchanged and click **Next**.

7. In this step, multiple EJB modules could be dispersed over available Application Servers. As we have only one EJB module, we leave this page unchanged and click **Next**.
8. This panel should show the group to role mapping defined during the application assembly. No changes are necessary, so click **Next**.
9. The current panel allows for apply general access rules to all bean methods that have not been protected during application assembly. You may consider attaching one of the available roles (and group) to all unprotected mean methods. After this, click **Next**.
10. In the last step, examine the summary and click **Finish** if you are satisfied.

    The deployment tool then goes through several steps, which it logs on the screen, to install the application. After successful completion the configuration has to be saved. The ear file will be installed in the location you specify during installation. If nothing is specified, it is installed in the default location <install root of WebSphere>/AppServer/installedApps.
11. Go to **Applications** -> **Redbook_Sample** -> **Redbook_Sample_EJB** -> **Connector Modules** -> **Redbook_Sample_Connector.rar** -> **J2C Connection Factories**. Enter `redbook_jni_connection` for the name and `redbook_jni_connection` for the JNDI name. Click **OK**.
12. As the final step, go back to **Applications** on the left side of the console window, expand the '+' and click on **Enterprise Applications**. In the list of the deployed applications that appears in the right part of the console window, you will see Redbook_Sample. Check its checkbox and click the **start** button on top of the list. If the application starts successfully, it is ready to be used.

Before running the application, copy the shared library directory_jni into a suitable location. Usually this is a directory in the PATH for Windows platforms or in a directory on the LIBPATH for AIX. Refer to section 10.4.5, "Java Native Interface" on page 238 for details about creating the shared library.

10.3.9 Running the application client

The application client is run like any other CORBA C++ client. Refer to section 10.3.4, "Configuring the application client" on page 222 for configuration details.The corba_client program takes up to two optional parameters, employee name and function code. For example: `corba_client -function 0 -employee john_doe` looks up the department name of the employee `john_doe`.

10.4 Revised application discussion

This section explains the revised application and some new concepts that are incorporated. Among other differences to be explained, the most important new concepts are:

- The use of enterprise bean wrappers for enclosing the application code in C
- The lack of authentication and authorization code (because this is done by WebSphere security)
- The use of JCA and JNI

10.4.1 Enterprise bean wrappers

The first task in the migration is to create an enterprise bean wrapper that encloses the server-side C program. This wrapper is needed to enable the C-based server business logic to execute within a Java based application server.

The wrapper is composed of a stateless session bean. For any basic enterprise bean, two Java interface classes must be specified for the enterprise bean: the home interface and the enterprise bean remote interface.

The requirements for the enterprise bean home interface are specified in the J2EE 1.3 specifications. However, we will be using the simplest form of enterprise bean, a stateless session bean. As a result the simple home interface will look like Example 10-6.

Example 10-6 Enterprise bean home interface

```
public interface DirectorySessionHome extends javax.ejb.EJBHome {
   public com.ibm.redbook.scenario3.EJB.DirectorySessionBean create()
      throws javax.ejb.CreateException, java.rmi.RemoteException;
}
```

The enterprise bean remote interface is based on the business logic and performs the same function as the DCE IDL file. The remote interface to be used in this scenario is shown in Example 10-7.

Example 10-7 Enterprise bean remote interface

```
package com.ibm.redbook.scenario3.EJB;
import java.rmi.RemoteException;
import javax.ejb.EJBObject;
```

```
public interface DirectorySessionBean extends EJBObject {
   String getDepartment(String emp_name) throws RemoteException;
   int getGrade(String emp_name)throws RemoteException;
   double getSalary(String emp_name) throws RemoteException;
}
```

In the next section we use these two interfaces to develop a CORBA client that calls the application server inside the WebSphere Application Server. The actual implementation of the remote interface is then discussed in section 10.4.4, “Application server” on page 236.

10.4.2 CORBA IDL file

From the definition of the two enterprise bean interface classes in the previous section, it is now possible to create two CORBA IDL files for use by the CORBA client. This is done with the **rmic** command using the idl option. For example, in the case of this example, these commands are:

```
rmic -idl DirectorySessionHome and
rmic -idl -classpath <was install root>/appserver/lib/j2ee.jar \
  DirectorySessionBean
```

For the case of the `DirectorySessionBean`, you may specify an optional `classpath` for the command as the Java class `DirectorySessionBean` imports the enterprise object class, which is located in the j2ee.jar archive.

The output of these commands is two header IDL files and two C++ program files named:

- DirectorySessionHome.idl
- DirectorySessionBean.idl
- DirectorySessionHome_C_stub.cpp
- DirectorySessionBean_C_stub.cpp

10.4.3 Application client

The two IDL files created in the previous step are used to build the application client. First, the IDL files must be converted into C program code. This is done with the idlc compiler.

One useful syntax for the idlc compiler command is:

```
idlc -euc:hh -I <include files> -d <target directory> <input file>
```

For this scenario, the idl commands used were:

```
idlc -euc:hh -I <was install root>/appserver/include DirectoryServerHome.idl
idlc -euc:hh -I <was install root>/appserver/include DirectoryServerBean.idl
```

These commands create four files: two C++ header files and and two C++ program files:

- DirectorySessionHome.hh
- DirectorySessionBean.hh
- DirectorySessionHome_C.cpp
- DirectorySessionBean_C.cpp.

A minor inconvenience in this process is that the generated C++ code for the home interface refers to the header file for the remote interface. However, the idlc compiler qualifies the reference with the full package name, if any, of the Java interface classes. As most Java programmers use a well-defined package structure during development, you may need to copy the header files from the idlc compiler to appropriate directories before compiling the C++ code, or else the idlc command for the remote idl can be re-executed with a target directory of com/ibm/redbook/sceanrio3/ejb.

The final step to completing the application client is to implement the application client business logic. As in the initial application, the application client will be split into two parts: initialization and business/presentation logic. The initialization code is completely different, as shown in Example 10-8.

Example 10-8 Client initialization code

```
int main(int argc, char *argv[]) {
   CORBA::Object_ptr objPtr;
   com::ibm::redbook::scenario3::EJB::DirectorySessionHome_ptr liptr;
   CORBA::ORB_ptr orbPtr = CORBA::ORB_init ( argc, argv, "DSOM" );
   if ( CORBA::is_nil(orbPtr) )
   {
      cerr << "Error initializing the ORB!" << endl;
      return -1;
   }
   try
   {
      Properties *props = CORBA::ORB::get_properties();
      const char *boothost = props->getProperty(
         "com.ibm.CORBA.bootstrapHostName"
      );
      const char *bootport=props->getProperty(
         "com.ibm.CORBA.bootstrapPort"
      );
      const char *servername=props->getProperty(
         "com.ibm.websphere.serverName"
      );
      const char *ejbname=props->getProperty("com.ibm.websphere.EJBName");
      char *home_url=(char *)malloc(512);
      strcpy(home_url,"corbaname::");
```

```
      if(boothost==NULL||bootport==NULL||servername==NULL||ejbname==NULL) {
         printf("Missing property in WASPROPS file\n");
         exit(1);
      }
      strcat(home_url,boothost);
      strcat(home_url,":");
      strcat(home_url,bootport);
      strcat(home_url,"/NameService#nodes/");
      strcat(home_url,boothost);
      strcat(home_url,"/servers/");
      strcat(home_url,servername);
      strcat(home_url,"/");
      strcat(home_url,ejbname);
      objPtr = orbPtr->string_to_object(home_url);
      liptr=com::ibm::redbook::scenario3::EJB::DirectorySessionHome\
         ::_narrow( objPtr);
      bptr=liptr->create();
   }
   catch (CORBA::UserException &ue)
   {
      cerr << "Caught a User Exception: " << ue.id() << endl;
      return -1;
   }
   catch (CORBA::SystemException &se)
   {
      cerr<<"Caught a System Exception: "<<se.id()<< ": "<<se.minor()<<endl;
      return -1;
   }
   execute_query(argc,argv);
   CORBA::release (bptr);
   CORBA::release ( orbPtr );
   return(0);
}
```

The differences arise because the coding language has changed from C to C++ and because the CORBA client object request broker does most of the system initialization work for us.

The business and presentation logic is shown in Example 10-9.

Example 10-9 Business and presentation logic

```
com::ibm::redbook::scenario3::EJB::DirectorySessionBean_ptr bptr;
void show_usage(int argc, char *argv[]) {
   printf("Usage is: %s <-function {0|1|2}> <-employee emp_name>",argv[0]);
   exit(1);
}
void execute_query(int argc, char **argv) {
   int i=0;
```

```
int query_type=0;
enum query_type {DEPT,GRADE,SALARY};
CORBA::WStringValue  *corba_name=com::ibm::ws::VtlUtil::toWStringValue("");
char *emp_name="";
CORBA::Long grade;
CORBA::Double salary;
CORBA::WStringValue *dept_name;
for(i=0;i<argc;i++) {
   if(strcmp(argv[i],"-function")==0) {
      if(i<argc-1) {
         query_type=atoi(argv[i+1]);
         i++;
      }
   }
   if(strcmp(argv[i],"-employee")==0) {
      if(i<argc-1) {
         emp_name=argv[i+1];
         corba_name=com::ibm::ws::VtlUtil::toWStringValue(argv[i+1]);
         i++;
      }
   }
   if(strcmp(argv[i],"-?")==0) show_usage(argc,argv);
}
switch(query_type) {
   case DEPT: {
      dept_name=bptr->getDepartment(corba_name);
      printf (
         "Department for %s is: %s\n",
         strcmp(emp_name,"")!=0?emp_name:"current user",
         com::ibm::ws::VtlUtil::WStringValueToString(dept_name)
      );
      break;
   }
   case GRADE: {
      grade=bptr->getGrade(corba_name);
      if(grade<0) {
         printf (
            "Grade for %s is: unknown\n",
             strcmp(emp_name,"")!=0?emp_name:"current user",
            grade
         );
      }
      else {
         printf (
            "Grade for %s is: %i\n",
             strcmp(emp_name,"")!=0?emp_name:"current user",
            grade
         );
      }
```

```
          break;
       }
       case SALARY:{
          salary=bptr->getSalary(corba_name);
          if(salary<0) {
             printf (
                "Salary for %s is: unknown\n",
                strcmp(emp_name,"")!=0?emp_name:"current user",
                salary
             );
          }
          else {
             printf (
                "Salary for %s is: %8.2f\n",
                strcmp(emp_name,"")!=0?emp_name:"current user",
                salary
             );
          }
          break;
       }
       default: {
          printf("Invalid query type %d\n",query_type);
          show_usage(argc,argv);
          break;
       }
    }
}
```

The program flow is almost identical to the DCE example. The differences that exist arise from the change in programming language from C to C++ and in the mapping of data types between C++ and CORBA.

The mapping complexity is the mapping of data types between CORBA and C++. CORBA supplies obvious mappings for simple types such as CORBA::Double for a C++ double type. However, more-complex objects such as C++ character strings or custom objects need special treatment and may require implementing a custom value type. In particular, the C++ character string type is mapped to the CORBA WStringValue type. Conversion from this type to something that can be manipulated with C++ string functions is done with the IBM VtlUtil utility class. In our example we use this class to create WStringValue and convert them to C++ strings. A more complete description of this process can be found in the IBM WebSphere Application Server Enterprise Edition Version 5 document collection book *IBM WebSphere Application Server Enterprise, Version 5, Common Object Request Broker Architecture (CORBA)*.

10.4.4 Application server

Finally, we come to the implementation of the replacement server. As discussed previously, the application server now is a stateless session bean hosted inside a WebSphere Application Server. The remote interface definition for the enterprise bean was described in section 10.4.1, “Enterprise bean wrappers” on page 230, and obviously is the basis for this code. The goal of this enterprise bean is to wrap the calls to our DCE application server code. This wrapper is relatively straightforward, and the Java code to implement the interface is shown in Example 10-14 on page 243.

Most of this code uses a default implementation of the empty method. The interesting thing in this code is that it uses a special class called Connection to bridge between the JNI Java and C worlds. The purpose of this class is discussed in sections 10.4.5, “Java Native Interface” on page 238 and 10.4.6, “J2EE Connector Architecture” on page 241.

With every enterprise bean there comes a deployment descriptor, which is used to inform the application server of the requirements of the bean. In particular, it is used to set the security policy. Example 10-10 shows the deployment descriptor for our bean that implements our security policy.

Example 10-10 Deployment descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 
2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar id="ejb-jar_ID">
   <display-name>Redbook Sample EJB</display-name>
   <enterprise-beans>
      <session id="Directory">
         <ejb-name>Directory</ejb-name>
         <home>com.ibm.redbook.scenario3.EJB.DirectorySessionHome</home>
      <remote>com.ibm.redbook.scenario3.EJB.DirectorySessionBean</remote>
<ejb-class>com.ibm.redbook.scenario3.EJB.DirectorySessionBeanImpl</ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Container</transaction-type>
         <resource-ref id="ResourceRef_1054820836658">
            <description></description>
            <res-ref-name>redbook_jni_connection</res-ref-name>
            <res-type>javax.resource.cci.ConnectionFactory</res-type>
            <res-auth>Application</res-auth>
            <res-sharing-scope>Unshareable</res-sharing-scope>
         </resource-ref>
      </session>
   </enterprise-beans>
   <assembly-descriptor>
```

```
<security-role>
   <description>Directory Manager</description>
   <role-name>manager</role-name>
</security-role>
<security-role>
   <description>Directory Employee</description>
   <role-name>employee</role-name>
</security-role>
<method-permission>
   <role-name>manager</role-name>
   <method>
      <ejb-name>Directory</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>getGrade</method-name>
      <method-params>
         <method-param>java.lang.String</method-param>
      </method-params>
   </method>
   <method>
      <ejb-name>Directory</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>getSalary</method-name>
      <method-params>
         <method-param>java.lang.String</method-param>
      </method-params>
   </method>
</method-permission>
<method-permission>
   <role-name>employee</role-name>
   <method>
      <ejb-name>Directory</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>getGrade</method-name>
      <method-params>
         <method-param>java.lang.String</method-param>
      </method-params>
   </method>
</method-permission>
<method-permission>
   <unchecked />
   <method>
      <ejb-name>Directory</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>getDepartment</method-name>
      <method-params>
         <method-param>java.lang.String</method-param>
      </method-params>
   </method>
</method-permission>
```

```
   </assembly-descriptor>
</ejb-jar>
```

10.4.5 Java Native Interface

The J2SE specification describes the Java Native Interface (JNI), which is a mechanism to execute code from other languages inside the Java Virtual Machine (JVM). JNI consists of two parts: a special class that describes the calls in the other language in terms of Java and a library that implements the calls.

Implementing the required Java class is fairly obvious and follows from the definition of the call in the native language. Example 10-11 shows a portion of the JNIConnection class of the resource adapter.

Example 10-11 Java specifying class

```
package com.ibm.redbook.scenario3.RA;
public class JNIConnection implements Connection,Serializable {

   static public native String getDepartmentJNI(String emp_name);
   static public native int getGradeJNI(String emp_name);
   static public native double getSalaryJNI(String emp_name);
   static {
      System.loadLibrary("directory_jni");
   }

   private JNIManagedConnection manconn;
   public String getDepartment(String emp_name) {
   return getDepartmentJNI(emp_name);
   }
   public int getGrade(String emp_name) {
   return getGradeJNI(emp_name);
   }
   public double getSalary(String emp_name) {
      return getSalaryJNI(emp_name);
   }
   public JNIConnection(JNIManagedConnection mc) {
      manconn = mc;
   }
   public Interaction createInteraction() throws NotSupportedException {
      throw new NotSupportedException("JNIConnection");
   }
   public void close() {
   }
   public LocalTransaction getLocalTransaction()
      throws NotSupportedException {
      throw new NotSupportedException("JNIConnection");
   }
```

```
   public ConnectionMetaData getMetaData() throws NotSupportedException {
      throw new NotSupportedException("JNIConnection");
   }
   public ResultSetInfo getResultSetInfo() throws NotSupportedException {
      throw new NotSupportedException("JNIConnection");
   }
}
```

In this case we tell Java that we will use three native functions: getDepartmentJNI, getGradeJNI, and getSalaryJNI. The names of these functions are not the same as those used in the DCE application server business logic and contained inside the shared library. Instead, these are pseudo function names constructed by Java.

The Java code System.loadLibrary("directory_jni") instructs the JVM to look for these special functions in a shared library called directory_jni.

From this class definition we can generate a C header file to be used in our implementation of these functions. This is done with the **javah** utility in the Java Development Kit. For this scenario, the **javah** command used was:

```
javah com.ibm.redbook.scenario3.RA.JNIConnection
```

This produced a single header file with the name com_ibm_redbook_scenario3_RA_JNIConnection.h

Example 10-12 shows the contents of this file.

Example 10-12 JNI Connection header file

```
#include <jni.h>
/* Header for class com_ibm_redbook_scenario3_RA_JNIConnection */
#ifndef _Included_com_ibm_redbook_scenario3_RA_JNIConnection
#define _Included_com_ibm_redbook_scenario3_RA_JNIConnection
#ifdef __cplusplusextern "C" {
#endif
JNIEXPORT jstring
   JNICALL Java_com_ibm_redbook_scenario3_RA_JNIConnection_getDepartmentJNI
   (
      JNIEnv *,
      jclass,
      jstring
   );
JNIEXPORT jint JNICALL
   Java_com_ibm_redbook_scenario3_RA_JNIConnection_getGradeJNI
   (
      JNIEnv *,
      jclass,
      jstring
```

```
   );
JNIEXPORT jdouble JNICALL
   Java_com_ibm_redbook_scenario3_RA_JNIConnection_getSalaryJNI
   (
      JNIEnv *,
      jclass,
      jstring
   );
#ifdef __cplusplus
}
#endif
#endif
```

The important details in this file are the names and prototypes for the C functions that will be called by the JVM.

To connect the native C business logic from the DCE example with this file, we must implement these functions in the directory_ini shared library, as shown in Example 10-13.

Example 10-13 Connection to C business logic

```
#include <stdio.h>
#include <string.h>
#include "com_ibm_redbook_scenario3_RA_JNIConnection.h"
JNIEXPORT jstring JNICALL
   Java_com_ibm_redbook_scenario3_RA_JNIConnection_getDepartmentJNI
   (
      JNIEnv *env,
      jobject obj,
      jstring s
   )
{
   const char *name;
   char *n;
   printf("In JNI Impl\n");
   name=(*env)->GetStringUTFChars(env,s,0);
   printf("In JNI Impl\n");
   n=getDepartmentImpl((char *)name);
   (*env)->ReleaseStringUTFChars(env,s,name);
   return (*env)->NewStringUTF(env,n);
}
JNIEXPORT jint
   JNICALL Java_com_ibm_redbook_scenario3_RA_JNIConnection_getGradeJNI(
      JNIEnv *env,
      jobject obj,
      jstring s
)
```

```
{
   const char *name=(*env)->GetStringUTFChars(env,s,0);
   const int g=getGradeImpl((char *)name);
   (*env)->ReleaseStringUTFChars(env,s,name);
   return (jint) g;
}
JNIEXPORT jdouble
   JNICALL Java_com_ibm_redbook_scenario3_RA_JNIConnection_getSalaryJNI(
      JNIEnv *env,
      jobject obj,
      jstring s
)
{
   const char *name=(*env)->GetStringUTFChars(env,s,0);
   const double sal=getSalaryImpl((char *)name);
   (*env)->ReleaseStringUTFChars(env,s,name);
   return (jdouble) sal;
}
```

As with the CORBA client, the complexity with the implementation comes with the mapping of Java objects to C data types. Like CORBA, Java provides mapping for simple types, for example jdouble for C's double. In the case of the Java string types the conversion requires methods from the Java JNIEnv object.

The next step for the application server is to compile the code in directory_jni_impl.c into a shared library called directory_jni.

The last step is to create a J2EE Enterprise Archive (EAR) with an EJB Archive containing the class DirectorySessionHome, DirectorySessionBean, and DirectorySessionBeanImpl, along with a deployment descriptor for deployment into a WebSphere Application Server.

At last we have arrived at the original C server business logic from the DCE example. However, if you put all of this code together, follow the WebSphere configuration steps and run the application client, the application server JVM will probably crash.

The reason for this is that enterprise beans cannot directly execute native code using JNI. To complete the solution requires exploring one more piece of the J2EE architecture landscape: the J2EE Connector Architecture (JCA).

10.4.6 J2EE Connector Architecture

The J2EE Connector Architecture is described in the specification document from Sun Microsystems. JCA is an attempt to standardize the way in which a J2EE Application Server communicates with resources such as database

managers, e-mail, messaging systems, and mainframe applications, known as Enterprise Information Systems (EIS). These systems typically are connection-oriented, and they operate by sending and receiving records in a well-scripted dialog. However, creating and destroying the connections to an EIS may take a long time and use a lot of other resources. Therefore, JCA enables EIS connections to be pooled, allowing their re-use, and shared, allowing many users of the connection simultaneously.

The JCA describes five main interfaces that must be implemented: *Connection*, *ConnectionFactory*, *ManagedConnection*, *ManagedConnectionFactory*, and *ConnectionManager*. The first two interfaces represent the end-user view of the EIS connection, and the next two represent the physical implementation of the connection. The ConnectionManager interface is usually implemented by an application server and is the interface responsible for pooling connections. This collection of classes is collectively known as a resource adapter.

To an end user, using JCA involves locating a suitable ConnectionFactory and calling the getConnection method to get the connection. All communication with an EIS then takes place through the methods defined by the Connection object.

For our needs, the JCA provides many features. Our immediate problem, however, is to allow Java Native Interface code to be executed by an enterprise bean. The JCA enables us to do this because it makes very few rules about the nature of the enterprise systems it is communicating with. One of the requirements is that classes in a resource adapter should be able to make a call involving JNI. Therefore, to make our server-side C code work through JNI, we must make it look like a JCA resource adapter.

The JCA specification requires eight classes to be implemented:

- JNIConnectionFactory (implements ConnectionFactory)
- JNIConnectionManager (implements ConnectionManager)
- JNIConnectionMetaData (implements ConnectionMetaData)
- JNIManagedConnectionFactory (implements ManagedConnectionFactory)
- JNIManagedConnection (implements ManagedConnection)
- JNIManagedConnectionMetaData (implements ManagedConnectionMetaData)
- JNIResourceAdapterMetaData (implements ResourceAdapterMetaData)
- JNIConnection (implements Connection)

Appendix C, “Scenario 3: Source code listings” on page 363 contains all of the code that is required to implement a simple resource adapter for our scenario. The JNIConnection class, shown in Example 10-14 on page 243, is of most

interest because it is responsible for making the calls to our server-side business logic. Compare the new implementation with the code from section 10.4.5, “Java Native Interface” on page 238.

Example 10-14 JNIConnection class

```
package com.ibm.redbook.scenario3.RA;

import java.io.PrintWriter;
import java.io.Serializable;
import java.util.Vector;

import javax.resource.NotSupportedException;
import javax.resource.ResourceException;
import javax.resource.cci.Connection;
import javax.resource.cci.ConnectionMetaData;
import javax.resource.cci.Interaction;
import javax.resource.cci.LocalTransaction;
import javax.resource.cci.ResultSetInfo;
import javax.security.auth.Subject;

public class JNIConnection implements Connection,Serializable {
   static public native String getDepartmentJNI(String emp_name);
   static public native int getGradeJNI(String emp_name);
   static public native double getSalaryJNI(String emp_name);
   static {
      System.loadLibrary("directory_jni");
   }

   private JNIManagedConnection manconn;
   public String getDepartment(String emp_name) {
      return getDepartmentJNI(emp_name);
   }
   public int getGrade(String emp_name) {
   return getGradeJNI(emp_name);
   }
   public double getSalary(String emp_name) {
      return getSalaryJNI(emp_name);
   }
   public JNIConnection(JNIManagedConnection mc) {
      manconn = mc;
   }
   public Interaction createInteraction() throws NotSupportedException {
      throw new NotSupportedException("JNIConnection");
   }
   public void close() {
   }
   public LocalTransaction getLocalTransaction()
      throws NotSupportedException {
```

```
      throw new NotSupportedException("JNIConnection");
   }
   public ConnectionMetaData getMetaData() throws NotSupportedException {
      throw new NotSupportedException("JNIConnection");
   }
   public ResultSetInfo getResultSetInfo() throws NotSupportedException {
      throw new NotSupportedException("JNIConnection");
   }
}
```

If you were to implement a resource adapter for your specific code, the only class that you should need to change is JNIConnection.

Note several points about this class. First, the enterprise bean wrapper DirectorySessionBeanImpl locates the Connection object through a ConnectionFactory class that is registered in JNDI and located through standard J2EE methods. The wrapper, as a stateless session bean, cannot cache a copy of the factory class, therefore it must rely on the capability of WebSphere to cache JNDI lookups to improve performance. Second, the JCA describes the communication flow between application server and enterprise information system in terms of records and an interaction. However, the specification also permits the development of custom methods in addition to the Connection interface to implement the interaction. We have done this for our example.

The final aspect of using a resource adapter is the way it gets deployed. Custom-purpose resource adapters, such as ours, usually are deployed as part of the same EAR that was used in section 10.4.4, “Application server” on page 236. The resource adapter is packaged as a separate module with its own deployment descriptor. Because it is bundled in the EAR, the resource adapter will be deployed along with EJB module.

Before the application is started, the JNDI name of the resource adapter used in the enterprise bean and the name used in the resource adapter module must be tied together. For the enterprise bean, the name that is looked up in JNDI gets mapped to a real name at deployment time. The name to be used for that is the one selected when the J2C ConnectionFactory for the resource adapter is configured.

The name is configured by using the Administrative Console and going to **Enterprise Applications** -> **(your application name)** -> **Connector Modules>** -> **(your connector name)** -> **Resource Adapter** -> **J2C Connection Factories** -> **(your connection factory)**. Some additional properties, such as Native Path for the resource adapter, can also be set.

10.5 Administration considerations and interfaces

The sample scenario in this chapter uses user and group information for authentication and authorization, where users and groups are migrated from DCE to the LDAP directory. Using an SSL certificate, a user in the revised application is authenticated by a comparison of the Common Name (CN) in the subject field of the certificate with an entry in the LDAP directory during the authentication procedure. On the other side, authorization decisions are proceeded due to the group membership of the client.

Administration tasks for the scenario include creation and deletion of user entries and modifying group membership information. In the migration environment, the user and group management can be done with well-known DCE administration tools, such as **dcecp** or **rgy_edit**. In addition, and after the removal of DCE, other LDAP directory administrative tools can be used. In the case of the IBM Directory Server, a Directory Management Tool (DMT) can be used.

Passwords needed for system administration accounts for IBM WebSphere Application Server are also kept in the LDAP directory. Password management can use the same administrative tools as for the J2EE application user. After the completion of the migration process, all administrative tasks concerning user and group management are done via LDAP directory management tools.

Two administration interfaces exist for managing deployed applications in IBM WebSphere Application Server and the IBM WebSphere Application Server:

- The WebSphere Administrative Console is a Web-based administration tool that provides the necessary functionality to deploy and manage applications, configure the security, and manage users and groups.
- wsadmin is a command line interface tool that can be used for scripting.

Besides the administrative console and the wsadmin tool, several scripts are provided to start, stop, and monitor IBM WebSphere Application Server processes.

Items of the Administrative console needed for the scenario

Mainly three items are used for the scenario:

- Applications: Use to install the example application or to stop and start deployed applications. To install new applications, navigate on the administrative console to **Applications** -> **Install New Application**. The installation procedure is described in detail in section 10.3.8, “Deploying and starting the application” on page 227. Navigate on the administrative console to **Applications** -> **Enterprise Applications** to administrate the applications.

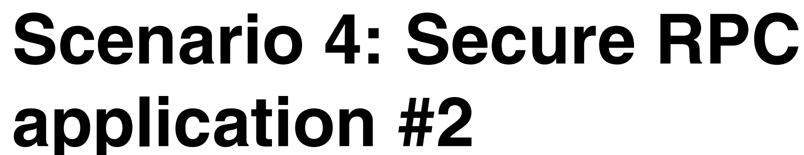
- Security: Used to set up the global security configuration of the IBM WebSphere Application Server. The security configuration comprises the authentication mechanism configuration, settings for SSL, configuration of access to the LDAP directory, and the CSIv2 protocol configuration. The necessary steps are described in detail in "Configuring WebSphere Application Server security" on page 212.
- System Administration: Assigns user and groups to administrative roles, as described in the *Administration interface* section of "Installing and configuring IBM WebSphere Application Server" on page 211. If the user registry resides in an LDAP directory, provide the Distinguished Name for the administrative account, instead of its common name, while assigning a user to an administrative role.

10.6 Discussion and conclusions

This scenario showed an application that uses DCE's secure remote procedure calls and how it could be moved to a Java-based application server with a significant re-use of the legacy C code. Although in this scenario the proportion of new code was relatively high in relation to existing DCE code, most real-life applications can be revised with considerably less new code.

The scenario also demonstrated both an easy path to enable legacy DCE applications for the Web, and an easy path for legacy DCE applications to migrate an entire application to Java. In the first situation we used a CORBA client to replace an existing DCE client, but we could have used a Web client such as a browser or applet. In the second situation, we wrote the enterprise bean wrapper layer to route calls either into newly written code or to our existing C code with little extra effort.

11



Scenario 4: Secure RPC application #2

This chapter presents scenario 4, which demonstrates Java strategies for an application that directly depends on the DCE RPC basics, RPC naming, RPC endpoint, RPC security, login, authorization, and PAC services. The strategies are:

- Re-code the entire application in Java.
- Replace DCE with the following services provided by WebSphere: RMI over IIOP, JNDI, LTPA, role-based authorization, CORBA CSIv2, and SSL with client certificates.
- Migrate the DCE registry database to IBM Directory Server, and configure WebSphere to use the migrated data.

Before reading this chapter, be sure to read Chapter 7, “Common replacement considerations” on page 111.

© Copyright IBM Corp. 2003. All rights reserved.

11.1 Scenario description

This scenario shows how an application that uses secure DCE RPC can be replaced by an application that uses IBM WebSphere Application Server. The description emphasizes how DCE security features and RPC services can be mapped to the corresponding mechanisms in a WebSphere environment. The chapter contents is as follows:

- The first section includes a discussion of the initial application with its DCE dependencies and how these dependencies are removed.
- The initial application with its DCE dependencies is detailed in section 11.2, “DCE application” on page 251.
- Section 11.3, “Replacement roadmap” on page 251 contains a description of the steps that are necessary to build and run the revised application.
- Section 11.4, “Revised application discussion” on page 259 explains and discusses the revised application.
- Some consideration for the management of the new environment is contained in section 11.5, “Administration considerations and interfaces” on page 262.
- Section 11.6, “Discussion and conclusions” on page 262 concludes the chapter.

11.1.1 Initial application with DCE dependencies

This sample uses the same DCE application that is used in Chapter 10, “Scenario 3: Secure RPC application #1” on page 195, which includes more details. For the application business logic to be migrated, refer to the DCE example in Chapter 9, “Scenario 2: Non-secure RPC application” on page 171.

11.1.2 Revised application without DCE dependencies

The revised application is a J2EE application without DCE dependencies. It consists of a Java client accessing an enterprise application inside the IBM WebSphere Application Server. The enterprise application in this case comprises a single enterprise bean (a stateless session bean) that implements the application logic. The communication between the J2EE client and the enterprise application takes place in a secured manner, using RMI over IIOP secured with CSIv2 together with SSL.

The J2EE client connects to the enterprise bean by looking up the enterprise bean’s home interface through the JNDI interface of the WebSphere Application Server Naming Service. Using the object reference to the home interface, the

J2EE client can obtain the object reference to the enterprise bean's remote interface, which contains the bean's actual business methods.

IBM Directory Server is employed as the user registry to store the user information. It holds user and group data migrated from the DCE registry that is being reused in this scenario. Figure 11-1 shows the revised application in the J2EE environment as it relates to the replacement scenario.

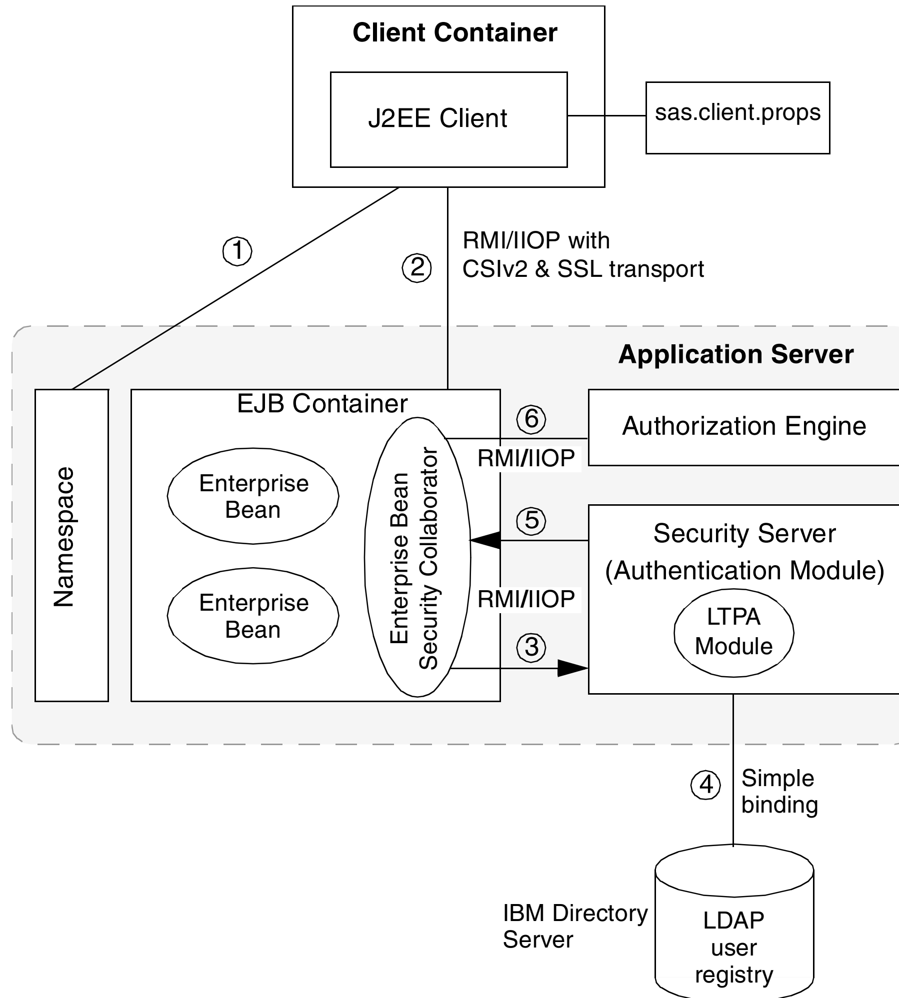


Figure 11-1 The revised application in the J2EE environment

The DCE application's login, authentication, authorization, RPC, and naming services have been replaced with their counterparts in J2EE environment. Upcoming sections describe each of them.

The steps involved when a J2EE client tries to access a secured enterprise application follow (numbers correspond to Figure 11-1 on page 249):

1. The J2EE client performs a JNDI lookup and receives the Interoperable Object Reference (IOR) of the enterprise bean.
2. The J2EE client sends authentication information (the client certificate sent via SSL) and requests access to the enterprise bean.
3. The enterprise bean Security Collaborator in the EJB Container sends authentication data to the Security Server and requests authentication.
4. The Security Server performs authentication by looking up the user in the LDAP user registry. In this scenario, this directory has been migrated from the DCE registry.
5. The Security Server returns the user credentials if this is a valid user.
6. The enterprise bean Security Collaborator performs authorization with the help of the Authorization Engine.

The Java client must be configured to access secured applications. Certain properties, such as the security settings of the client ORB, must be provided in a file called sas.client.props. The JVM (Java Virtual Machine) in which the WebSphere Application Client will run should be set to use this property file by adding the directive:

```
com.ibm.CORBA.ConfigURL=<location of the properties file>
```

WebSphere can also be configured to use external authorization systems such as IBM Tivoli Access Manager. For more information, refer to the IBM Redbook *IBM WebSphere V5.0 Security*, SG24-6573.

Authentication in the revised application

Authentication in this scenario uses client certificates, as in the previous scenario. See “Authentication in the revised application” on page 200.

Authorization in the revised application

In the revised application, authorization decisions are made in two places: first while accessing the EJB Container in order to invoke the enterprise bean instance, and second with the aid of user-written authorization code within the enterprise bean. To achieve this, two roles with different authorization options are defined. The authorization roles are labeled as users and managers. The execution path of the code depends on the security role of the user. During the deployment phase of the enterprise application, a mapping between the roles of the enterprise application and existing groups in IBM Directory Server is done. Hence it is possible to control the access to the application with the aid of group memberships equivalent to access control mechanisms in DCE.

Naming in the revised application

In the revised application, a single server configuration is used (one Application Server running on a single node) because this reflects many DCE applications. The J2EE client bootstraps to the host on which the Application Server is running. It uses the default initial context of the server root to bind to the name space of the Application Server. The JNDI binding used is a simple name, MirrorHome. The JNDI operations are performed with java: URL names. Names bound under these names are bound to a completely different name space, which is local to the calling process.

11.2 DCE application

The secure RPC portion of the application used in this scenario is the same as in the application used in the previous scenario (section 10.2, “DCE application” on page 203). The application logic used in the current scenario is identical to the application logic used and explained in section 9.2, “DCE application” on page 177. Refer to these sections, as the code and explanations are not repeated here.

11.3 Replacement roadmap

The roadmap for this replacement scenario includes:

- Meeting the software requirements
- Installing and configuring IBM WebSphere Application Server, which includes the security configuration and specifying LDAP filter rules
- Configuring WebSphere Application Server security
- Configuring the application client
- Developing the application
- Configuring IBM Directory Server
- Assembling, deploying, and starting the application
- Running the application client

Refer to section 10.3, “Replacement roadmap” on page 210 for the general roadmap that is common to the previous scenario and this scenario. The following sections only describe the tasks that are specific to the current scenario.

11.3.1 Software requirements

The software requirements for running the revised application run time are:

- IBM WebSphere Application Server, Version 5 (any edition)
- IBM Directory Server 3.2.2
- IBM Java JDK 1.3.1 (included with IBM WebSphere Application Server)

The software requirements for the revised application development are:

- IBM WebSphere Studio Application Developer 5.01 or other Java development tool

11.3.2 Installing and configuring IBM WebSphere Application Server

Refer to section 10.3.2, "Installing and configuring IBM WebSphere Application Server" on page 211, as the process in the previous scenario is the same for this scenario.

11.3.3 Configuring WebSphere Application Server security

Login, authentication, and authorization of the revised application are performed by WebSphere Application Server, whose security configuration is very similar with the one used in the previous scenario (section 10.3.3, "Configuring WebSphere Application Server security" on page 212), except that, to demonstrate more practical situation, this scenario uses certificates that are created by a third-party Certificate Authority (CA). For this reason, the preparation of the SSL key and trust files and the configuration of the LDAP filter rules differ from the previous scenario. For the LDAP filter rules, the subject CN in the certificates in this scenario exactly matches the krbPrincipalName attribute in the LDAP directory.

Prepare SSL key and trust files

The current scenario uses certificates created by a trusted Certificate Authority (CA), as described in section 7.7.7, "Using certificates from a Certificate Authority (CA)" on page 121. In brief, the following must be done:

1. Copy the default key and trust files provided with WebSphere to files that are used for the scenario. For example:
   - Copy DummyServerKeyFile.jks to ITSOSrvKeyFile.jks
   - Copy DummyServerTrustFile.jks to ITSOSrvTrustFile.jks
   - Copy DummyClientKeyFile.jks to ITSOClientKeyFile.jks
   - Copy DummyClientTrustFile.jks to ITSOClientTrustFile.jks

(On AIX, these default key files are in the /usr/WebSphere/AppServer/etc directory.)

2. With the **ikeyman** utility, import the root certificate of the CA into the application server's and client's trust files (ITSOSrvTrustFile.jks and ITSOClientTrustFile.jks).
3. Create certificate requests for the application client and server with the **ikeyman** utility using the key files for each. The subject CN of the application client's certificate in this scenario exactly matches the Kerberos principal name in the LDAP directory (cn=john_doe@realm1.austin.ibm.com, for example).

   Note that this is different from the previous scenario where the subject CN only included the DCE principal name (john_doe). In this scenario, the subject CN includes the DCE principal name and the realm name (john_doe@realm1.austin.ibm.com).
4. Send the certificate requests from the previous step to the CA in order to request the application client's and server's certificates.
5. Once issued by the CA, the certificates must be received (imported), with the application client's and server's **ikeyman** utilities, into their respective key files.

> **Note:** For these steps, use the **ikeyman** utility that is provided with WebSphere Application Server because it supports the jks file format. On AIX, launch **ikeyman** with /usr/WebSphere/AppServer/bin/ikeyman.sh

Configure LDAP filter rules

In the current scenario, the client certificates created by the CA are created in such a way that their subject CNs contain exactly the names as they are stored in the krbPrincipalName attributes of the LDAP directory. For example, the subject CN in a certificate is john_doe@realm1.austin.ibm.com and the krbPrincipalName is exactly the same, john_doe@realm1.austin.ibm.com. This differs from the previous scenario where the subject CN was only a part of the krbPrincipalName.

The LDAP filters for users and groups remain the same as in the previous scenario (refer to “Configure LDAP filter rules” on page 218), except that the Certificate Filter differs and is set to:

```
krbPrincipalName=${SubjectCN}
```

11.3.4 Configuring the application client

Apart from developing and assembling the application client as described in sections 11.3.5, “Developing the application” on page 254 and 11.3.7,

“Assembling the scenario application” on page 255, the only configuration required is to configure the sas.client.props file to supply authentication information for the user and to configure it to use client certificate and SSL. Here, we assume that you have already procured the client certificate and that the keystore and truststore files have been created. Refer to section 7.7, “SSL implementation hints” on page 116 for details on SSL and certificates. In the sas.client.props file, be sure the following properties are set to the values mentioned, as in Example 11-1. The remaining properties can keep the default values.

Example 11-1 Application client properties file (excerpt)

```
com.ibm.CORBA.securityEnabled=true
com.ibm.CSI.protocol=csiv2
com.ibm.CSI.performTLClientAuthenticationSupported=true
com.ibm.ssl.protocol=SSL
com.ibm.ssl.keyStoreType=JKS
com.ibm.ssl.keyStore=C:/Program Files/WebSphere/AppClient/etc/
  ITSOClientKeyFile.jks
com.ibm.ssl.keyStorePassword=WebAS
com.ibm.ssl.trustStoreType=JKS
com.ibm.ssl.trustStore=C:/Program Files/WebSphere/AppClient/etc/
  ITSOClientTrustFile.jks
com.ibm.ssl.trustStorePassword=WebAS
com.ibm.ssl.keyStoreClientAlias=<client certificate alias in the keystore file>
```

Note that some lines in Example 11-1 are too long to fit the column of this text, so they are shown in two lines. Refer to Appendix D, “Scenario 4: Source code listings” on page 403 for the properties file of the revised application client.

11.3.5 Developing the application

Read the general comments in the previous scenario on application development in section 10.3.5, “Developing the application” on page 223, where some differences between application development in a legacy DCE and a J2EE environments are briefly discussed. Another difference is the absence of IDL interface definitions in a pure J2EE environment. The home and remote interfaces of enterprise beans define the interfaces used for client/server communications. Java objects as method parameters or method results can be transferred by value as long as they are *serializable* objects. Only when interfacing with CORBA clients (see Chapter 10, “Scenario 3: Secure RPC application #1” on page 195) can an IDL definition be generated from the enterprise bean interfaces.

The output of the enterprise application development process for this scenario is two `jar` files, which in turn are input for the enterprise application assembling

process as described in section 11.3.7, “Assembling the scenario application” on page 255. The two `jar` files are:

- MirrorServer.jar: This contains the stateless session bean MirrorBean together with home (MirrorHome) and remote (Mirror) interface class files, as well as the class file for the MirrorException.
- MirrorClient.jar: This contains the J2EE application client’s class file MirrorClient.

11.3.6 Configuring IBM Directory Server

This scenario uses the same prerequisites for the IBM Directory Server and its configuration as in the previous scenario. (See section 10.3.6, “Configuring IBM Directory Server” on page 223.)

11.3.7 Assembling the scenario application

The following steps assemble the scenario application, called Mirror. For a general description of the application assembly process, read section 10.3.7, “Assembling the scenario application” on page 223.

On a system that has WebSphere Application Server installed and that has access to the jar files for the J2EE client and Mirror enterprise beans, do the following to assemble the application:

1. On Windows, start the WebSphere Application Server Application Assembly Tool by selecting **Start** -> **Programs** -> **IBM WebSphere** -> **Application Server v5.0** -> **Application Assembly Tool**.

   On Unix, start the Application Assembly Tool using this shell script:

   ```
   <install root of WebSphere>/AppServer/bin/assembly.sh
   ```
2. Select **Application** from the choice of subjects that can be created.
3. In the panel for the new enterprise application, enter the name of the application (`MirrorApp`) and click **Apply**.
4. On the left side of the Assembly Tools window, select **EJB Modules** and right-click to select **New**.
5. Enter the common configuration information for the enterprise beans:
   a. Replace the default jar file name with `MirrorBean.jar`
   b. Enter the Display Name `MirrorBean`
   c. Click **Browse** to find the jar file for the J2EE client MirrorClient.jar provided by the application development process. Select the jar file and click **Open**.

6. Add the enterprise bean `Mirrorbean` to the created EJB Module and configure it:

   a. Under **EJB Modules**, expand the '+' on the entry **MirrorBean**, select **Session Beans** and right-click to select **New**.

   b. In the General tab of the New Session Bean window, enter `MirrorBean` as EJB Name, and browse and enter the EJB class, EJB home, and remote interface, by accessing the MirrorServer.jar file for the MirrorBean provided by the application development process. Enter `MirrorBean.class` for EJB Class, `MirrorHome.class` for Home and `Mirror` for Interface.

   c. Select the Advanced tab, and be sure that **stateless** is selected for Session Type.

   d. Select the Bindings tab and enter the JNDI name for MirrorBean, which in this case is `MirrorHome`.

   e. After this, select **OK** in the General tab.

7. On the left part of the assembly tool's main window, go back to application level MirrorApp and select **Files**. Right-click to select **Add Files**. Browse to find the jar file for MirrorBean, select **MirrorException.class**, and click **Add**. This makes the MirrorException class visible for the whole application.

8. Next, configure the J2EE client application for the MirrorBean enterprise application. For that purpose, select **Application Clients** in the left part of the assembly tool's main window and right-click to select **New**.

9. In the General tab of the New Application Clients window, enter `MirrorClt.jar` as Filename and `MirrorClient` as Display name. Click **Browse** to find and select the J2EE client's main class **MirrorClient.class** from the application client's jar file provided by the development process. Click **OK** in the Select file for Main class window to create the envelope for the J2EE client.

10. In the left part of the main window of the assembly tool, select **Application Clients**, expand the '+' on the entry of **MirrorClient** and select **EJB References**. Right-click on it and select **New**.

11. In the General tab of the New EJB Reference window, enter the name `MirrorHome`, which must map the JNDI name entered later. Then browse twice to access the MirrorBean's jar file to enter references to the enterprise beans home- and remote interface class files.

12. In the Bindings tab of the New EJB Reference window, enter the JNDI name to be used, which is `MirrorHome`.

13. Now, that the application client is assembled, return to the enterprise bean part to complete the security part of the application assembly. For this purpose, return to the application level on the left side of the main window, select **Security roles**, right-click to select **New**.

14.Create two new roles, `manager` and `user`. In the Bindings tab of the New Security Role window, bind the `manager` role with the `MirrorManagers` registry group and the `user` role with the `MirrorUsers` registry group. In the scenario example, these groups have been created by DCE and migrated to the LDAP directory. Please note that you must use the full Distinguished Name of the groups with the assembly tool. When this is done, the manager and user roles are known globally for the MirrorApp enterprise application and are ready to be used for further configuration.

15.Next, specify for the EJB Module MirrorBean which roles to use locally: Under **EJB Modules**, expand the '+' on **MirrorBean**, and right-click **Security roles** to select **`New`** twice, to create the roles `manager` and `user` here as well.

16.After this, a link has to be made from the local role definitions, which the MirrorBean will use, to the definitions known to the whole application. This is done on the level **MirrorApp** -> **MirrorBean** -> **Session Beans** -> **MirrorBean** -> **Security Role References**. Right-click to create two new links. Use the same name for **name** and **link** each time. The local role manager now maps to global role manager, and the local role user maps to global role user.

17.The configuration is ready to annotate enterprise bean methods with role-based permissions: From **MirrorApp** -> **MirrorBean** -> **Session Beans**, select **Method Permissions**, and right-click to create new permissions.

18.In the **New Method Permission** window, enter a method permission name. Click **Add** for Methods and select the method to be annotated. In our scenario, this is the reflect method of the enterprise bean MirrorBean. This method contains the business logic of our example. Add `reflect` and click **OK**. Next, click **Add** for Roles, and select the roles that shall have access to reflect. Click **OK**.

19.Go to **File** -> **Save As** and save the application archive as `MirrorApp.ear`.

20.Optional: As the next step of the application assembling process, go to **File** -> **Generate Code for Deployment** in the main window of the assembly tool to make the tool create the ear file needed for the application deployment.

21.Optional: In the Generate Code for Deployment window, click **Generate now**. The tool performs and logs several steps to create the ear file. If no error occurs, a file named Deployed_MirrorApp.ear is generated.

The last two steps are optional, as the deployment code can be generated during application deployment phase.

Exit the tool and continue with deployment as explained in the next section.

11.3.8 Deploying and starting the application

These steps deploy and start the Mirror enterprise application, the sample enterprise used in this scenario. A general description of the application deployment process can be found in section 10.3.8, "Deploying and starting the application" on page 227.

As a prerequisite, be sure that the Enterprise Application Archive file MirrorApp.ear is accessible, then open the WebSphere Administrative Console. Perform the following steps to deploy the MirrorApp application:

1. The left side of the Administrative Console shows a list of subjects that can be configured for each Application Server. In our example there is only one instance of an Application Server. Expand the '+' on **Applications** and click **Install New Application**.
2. On the right side of the console panel, enter the local or remote path to MirrorApp.ear and click **Next**.
3. Check **Generate general bindings** in the current panel and click **Next**.
4. Check **Deploy EJBs**. You may consider checking **Enable class reloading**, but this feature is not needed for the sample scenario. Click **Next**.
5. No further deployment options are necessary in the current panel. Click **Next**.
6. The current panel should already contain the JNDI Name MirrorHome for MirrorBean. Leave it unchanged and click **Next**.
7. In this step multiple EJB modules could be dispersed over available Application Servers. As we have only one EJB module, we leave this panel unchanged and click **Next**.
8. This panel should show the group to role mapping, which was defined during the application assembly. No changes are necessary, so click **Next**.
9. The current panel would allow for applying general access rules to all bean methods that have not been protected during application assembly. You may consider attaching one of the available roles (manager and group) to all unprotected mean methods. After this, click **Next**.
10. In the last step, examine the summary and click **Finish** if you are satisfied.

    The deployment tool then goes through several steps, which it logs on the screen, to install the application. After successful completion, save the configuration. The ear file will be installed in the location you specify during installation. If nothing is specified, it is installed in the default location <install root of WebSphere>/AppServer/installedApps.
11. As a last step, return to **Applications** on the left part of the console page, expand the '+' and click **Enterprise Applications**. In the list of the deployed applications, which then appears on the right side of the console window, you

should find MirrorApp. Check its checkbox and click the **start** button on top of the list. If the application starts successfully, it is ready to be used.

11.3.9 Running the application client

The J2EE client example is started using the **launchClient** tool that is included in the distribution of the WebSphere Application Server and WebSphere Application client product. The **launchClient** tool starts the J2EE client application in a WebSphere client container. The command is:

```
$ launchClient Deployed_MirrorApp.ear
```

If a different WebSphere Application Server should be used, the option -CCBootstrapHost=<server hostname> can be specified. For problem determination, the option -CCTrace=true turns out to be useful. Refer to WebSphere Application Server product documentation for more available options. During startup, the J2EE client application accesses the sas.client.properties file to obtain the security settings to be used. The property file used in this scenario can be found in Appendix D, “Scenario 4: Source code listings” on page 403. Using this property file and the key files the property file points to will let the J2EE client assume the identity of john_doe, which exists in the LDAP user registry. In this scenario john_doe is a DCE principal that has been migrated to the LDAP directory using the steps as described in section 8.3.2, “Migration of DCE security registry to IBM Directory Server” on page 139.

11.4 Revised application discussion

This section describes the Java code that makes up the application client and server in the new environment. It shows that only small portions of code are necessary to achieve this, a major reason being that almost all code for authentication, authorization, RPC, and naming, which had to be coded in DCE, is left to configuration in J2EE. (See section 11.3, “Replacement roadmap” on page 251.) An exception to this is the piece of programmatic authorization, which this example includes. (See “Programmatic security” on page 42.)

> **Note:** Declarative authorization should be preferred whenever possible, because it is achieved by configuration (during application assembling and deployment) instead of by programming. This results in more flexibility for the application with regard to authorization policy changes.

The application logic remains unchanged; the application server returns a string passed by the application client in reverse character order.

11.4.1 Application client

The MirrorClient Java application client gains access to the MirrorBean enterprise bean's remote interface. It then invokes the reflect method of this interface, passing a string that it had previously read from user input. After successful completion of the call, the MirrorClient prints out the returned result string. If the method call fails, the application client catches the raised exceptions and prints the information carried by them.

Example 11-2 details the code necessary for the application client to gain access to the MirrorBean's remote interface. To get access to MirrorBean's remote interface, the Mirror application client looks up the object reference to the home interface, MirrorHome, of the enterprise bean MirrorBean, using the JNDI interface to the WebSphere Application Server Name Service. This object reference is then narrowed down by the application client to a reference to the MirrorBean's home interface. PortableRemoteObject indicates that the CORBA mechanism is used to load the home object.

Example 11-2 Getting access to the remote interface of the enterprise bean

```
Context initial = new InitialContext();
Object obj = initial.lookup("java:comp/env/MirrorHome");
home = (MirrorHome)PortableRemoteObject.narrow(obj, MirrorHome.class);
mirror = home.create()
```

The J2EE client then calls the method *create*, which is exposed by the MirrorHome interface. After successful completion, this call returns a reference to the remote interface of the MirrorBean.

Once the application client has obtained an object reference to the MirrorBean's remote interface, it can begin to exchange application data with the application server. Note that no more security coding is necessary for authentication or authorization. The application client is aware of possible authentication or authorization problems only through exception handling.

11.4.2 Application server

Concerning the application server, the Java application programmer can concentrate on developing the business logic. The business logic in Example 11-3 on page 261 consists of reversing a string delivered by the application client and returning the string as a method result back to the application client.

As this example uses a stateless session bean, no further code for state preservation has to be provided. Little additional code is added to demonstrate programmatic authorization.

Example 11-3 Authorization in application code

```
public String reflect(String message) throws MirrorException {

      System.out.println("received message: " + message);
      //
      // example how authz can be performed in code
      //
      java.security.Principal principal = sessionContext.getCallerPrincipal();
      java.lang.String callerId = principal.getName();
      System.out.println("received call from: " + callerId);
      boolean isMgr = sessionContext.isCallerInRole("manager");
      boolean isUser = sessionContext.isCallerInRole("user");
      if(!isMgr) {
         if (isUser) {
            System.out.println("Sorry, " + callerId + " you are just a user");
            throw new MirrorException
               ("Sorry, " + callerId + " you are just a user");
         }
         //
         // Allow access for other roles, to keep security configurable
         //
      }
      //
      // perform the actual 'business logic'
      //
      StringBuffer tmp = new StringBuffer(message);
      return (tmp.reverse()).toString();
   }
}
```

This presents a simple example of the way authorization code in the application could look. In this example, it is assumed that the container where MirrorBean is deployed enables the reflect method to be called by the user and manager roles. However, inside the method implementation, the access rights are refined to enable access only through the manager role. User access is denied, and the denial is signaled back to the application client with MirrorException.

Note: In the Mirror example, the application programmer actually will write only the code just presented. All of the rest, including a template for the enterprise bean, the code for the home and remote interface, as well as the templates for the declared exceptions, are usually generated by an integrated development environment that is used for J2EE development. For WebSphere, this is WebSphere Studio Application Developer Version 5.

11.5 Administration considerations and interfaces

This scenario uses the same WebSphere security and configuration features as the previous scenario. Thus, the administration considerations and interfaces of the scenario described in section 10.5, “Administration considerations and interfaces” on page 245, are applicable to the scenario in this chapter.

11.6 Discussion and conclusions

Among the middleware products available today, J2EE Application Servers come closest to replacing DCE with a single technology. (See section 2.3.3, “DCE services that can be replaced by J2EE” on page 37.) DCE RPC, naming, security, failover, and workload balancing find powerful counterparts in the J2EE Application Server technology. However, moving from DCE to J2EE cannot be just a one-by-one replacement; rather, it involves a change of paradigm. The J2EE paradigm takes away the burden of middleware programming by introducing an extra layer between an application and the middleware, called a container. Figure 11-2 depicts that difference.

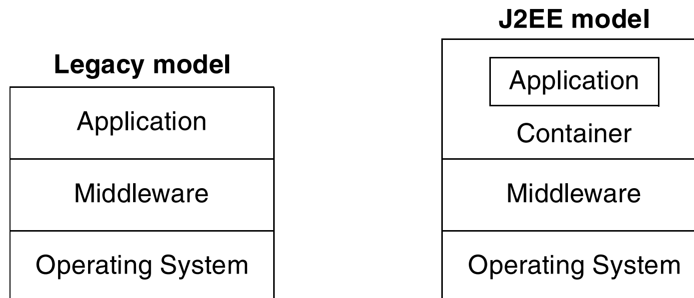


Figure 11-2 Traditional and J2EE application layering

In the J2EE model, the handling of the middleware technology is moved to the vendor of the Application Server. This has the advantage of greater productivity (at a high quality level) of the application’s development process because the middleware functionality is carried out underneath. A potential downside of this model is that J2EE Application Servers from different vendors may not be compatible at the middleware level. For example, clustering of enterprise beans may not be possible between application servers from different vendors. With DCE, vendor-independence is ensured as DCE is built on one common source code base by every vendor. Another downside may be a performance impact caused by the additional layer and the nature of the Java technology.

Where DCE primarily relies on the Kerberos security standard with fine-grained access control capabilities built on top, J2EE Application Servers today come with a diverse set of security protocols and features, which are either integrated with the product or pluggable. An example is the integration with public key technology, such as SSL, which has become a widely used industry standard. With the support for SSL, application servers also can be deployed as Web servers, exposed to intranets and the Internet.

Although securing a distributed application in J2EE is mostly achieved by configuration (rather than programming), the task is nevertheless not trivial. The available security features must be deployed in a proper way, such that the distributed applications get the security they really need. For this, it is important to understand the available security mechanisms and their configuration. Some J2EE Application Server environment might even support proprietary security mechanisms.

Given the advancements in security technology that are available to J2EE Application Servers, many concepts and features of DCE security can be found in the new technology as well. For example, the DCE security delegation *impersonation* feature can be compared with the J2EE *run-as* feature.

As an alternative to the strategy used in this scenario, Web services could serve as another replacement strategy for environments that use or migrate to Web-based clients. Web services is a fast-evolving technology that is worth evaluating in many replacement environments. Some versions of IBM WebSphere support Web services.

Appendixes

© Copyright IBM Corp. 2003. All rights reserved.

A

Scenario 1: Source code listings

This appendix lists the complete source code that was explained in pieces in Chapter 8, “Scenario 1: GSS-API application” on page 125. Specifically, this appendix contains:

- The source code of the client and server programs with DCE dependencies
- The source code of the revised client and server program without DCE dependencies
- The makefiles used for these programs
- Additional source code modules for authorization
- Header files
- Application configuration files

Refer to Appendix E, “Additional material” on page 417 for instructions for downloading the source code samples included in this appendix.

© Copyright IBM Corp. 2003. All rights reserved.

Application with DCE dependencies

We start with the source code for an application with DCE dependencies:

- s1.mak contains information about how the application is compiled and linked on the Windows operating system.
- Makefile contains information about how the application is compiled and linked on the AIX operating system.
- client_s1.c contains the client code. After acquiring an initial login context, the application client establishes a common security context with the server. Then the client exchanges application messages with the application server via socket communication.
- server_s1.c contains the server code. The application server registers itself and listens for connections. Then it accept a common security context with the client. If this is successful, it can exchange application messages with clients. Furthermore, it uses the dce_authz.c module to authorize the client in case of a request.
- dce_authz.c contains the code for the authorization module. The module has two major tasks: To set up an ACL on an object and to provide functions to handle requests for authorization decisions initiated by the server.
- utils_s1.c provides functions for socket communication. Furthermore, the task of encryption and decryption is handled in this module.
- utils_s1.h represents the header file for utils_s1.c.

Makefile for application client

Description: Makefile for Windows application client
File name: s1.mak

```
!include <NTWIN32.MAK>
!if "$(CPU)" == "ALPHA"
TARGET=alpha
SWITCHES=-DALPHA
!else
TARGET=intel
SWITCHES=-D_X86=1 -DM_I86
!endif

_LIBS_= \
   libdce.lib \
   pthreads.lib \
   wsock32.lib \
   $(conlibsdll)
```

```
CFLAGS= -I. $(cflags:-W3=) $(cvarsdll) $(SWITCHES) -Zi -Od -Gz -nologo

LINK = link $(ldebug) $(conlflags)

all: client_s1.exe

client : client_s1.exe

client_s1.exe : client_s1.obj utils_s1.obj dce_authz.obj
   $(LINK) /out:$@ $** $(_LIBS_)

dce_authz.obj: dce_authz.c
   $(CC) $(CFLAGS) dce_authz.c

client_s1.obj: client_s1.c
   $(CC) $(CFLAGS) client_s1.c

utils_s1.obj: utils_s1.c
   $(CC) $(CFLAGS) utils_s1.c

clean :
   del *.obj

clobber :
   del *.obj
   del *.exe
```

Makefile for application server

Description: Makefile for AIX application server
File name: Makefile

```
CC=/usr/vacpp/bin/xlC_r4

CFLAGS  -I. -g -o

all: server

server : server_s1.o utils_s1.o dce_authz.o
   $(CC) ${CFLAGS} server_s1 server_s1.o utils_s1.o dce_authz.o -ldce

server_s1.o :
   $(CC) -c  server_s1.c

utils_s1.o :
   $(CC) -c utils_s1.c
```

```
dce_authz.o :
   $(CC) -c dce_authz.c

clean :
   rm -f *.o

clobber :
   rm -f server_s1 *.o
```

DCE dependent application client

Description: Application client with DCE dependencies
File name: client_s1.c

```
#include <dce/sec_login.h>
#include <dce/gssapi.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include "utils_s1.h"

#undef write
#define write(fd, bufptr, buflen) send(fd, bufptr, buflen, 0)
#undef read
#define read(fd, bufptr, buflen) recv(fd, bufptr, buflen, 0)
#undef close
#define close(fd) closesocket(fd)

#define SERVICE_SECURE 2
#define value(x) (((x>='0') && (x<='9'))?x-'0':-1)
#define MAX_HOST_LENGTH 64

char * port_param = NULL;
char * host_param = NULL;
int port;
char port_name[512];
unsigned host_addr;
struct hostent * host_entry = NULL;

gss_ctx_id_t security_context;

/************************/
/* function login_to_dce */
int login_to_dce(
   unsigned_char_p_t prin_name,   /* server principal name.*/
   unsigned_char_p_t keytab){     /* keytab file */

   sec_passwd_rec_t         *keydata;
```

```
sec_login_auth_src_t     auth_src;
boolean32                reset_pwd;
sec_login_handle_t       login_context;
unsigned32               status = error_status_ok;

/* create a context and get the login context */
sec_login_setup_identity(
   prin_name,
   sec_login_no_flags,
   &login_context,
   &status);
if (status != error_status_ok){
   fprintf(stderr,
      "Error - sec_login_setup_identity(), status: %d\n",status);
   return -1;
}

/* get secret key from the keytab file */
sec_key_mgmt_get_key(
   rpc_c_authn_dce_secret,
   keytab,
   prin_name,
   0,
   (void**)&keydata,
   &status);
if (status != error_status_ok){
   fprintf(stderr,
      "Error - sec_key_mgmt_get_key(), status: %d\n",status);
   return -1;
}

/* validate the login context */
sec_login_validate_identity(
   login_context,
   keydata,
   &reset_pwd,
   &auth_src,
   &status);
if (status != error_status_ok){
   fprintf(stderr,
      "Error - sec_login_validate_identity(), status: %d\n",status);
   return -1;
}

/* finally, set the context */
sec_login_set_context(
   login_context,
   &status);
if (status != error_status_ok){
```

```
      fprintf(stderr, "Error - sec_login_set_context(), status: %d\n",status);
      return -1;
   }

   printf("DCE login successfull\n");
   return 0;
}

/**************************/
/* function define_service */
/* exchanges authentication token with server for mutual authentication */

int define_service(
   int socket,
   gss_ctx_id_t * sec_ctx,
   char * server_name){

   OM_uint32 maj_stat, min_stat;
   gss_name_t server;
   gss_buffer_desc name;
   gss_buffer_desc snd_token;
   gss_buffer_desc rcv_token;
   int done;
   unsigned char length_low, length_high;
   int length;
   name.length = strlen(server_name);
   name.value = server_name;

   maj_stat = gss_import_name(
               &min_stat,
               &name,
               GSS_C_NULL_OID,
               &server);
   if (GSS_ERROR(maj_stat)) {
      display_error("gss_import_name",
         maj_stat,
         min_stat);
         return -1;
   };
   *sec_ctx = GSS_C_NO_CONTEXT;
   rcv_token.length = 0;
   rcv_token.value = NULL;
   snd_token.length = 0;
   snd_token.value = NULL;
   done = 0;

   while (!done) {

   maj_stat = gss_init_sec_context(
```

```
                   &min_stat,
                   GSS_C_NO_CREDENTIAL, /* Default cred */
                   sec_ctx,
                   server,
                   GSSDCE_C_OID_DCE_KRBV5_DES,
                   GSS_C_MUTUAL_FLAG | GSS_C_INTEG_FLAG |GSS_C_INTEG_FLAG,
                   24*60*60,
                   GSS_C_NO_CHANNEL_BINDINGS,
                   &rcv_token,
                   NULL,
                   &snd_token,
                   NULL,
                   NULL);

   if (GSS_ERROR(maj_stat)) {
     display_error("gss_init_sec_context",
        maj_stat,
        min_stat);
        return -1;
   };

   if (snd_token.length != 0) {
      length_low = snd_token.length & 0xff;
      length_high = snd_token.length >> 8;
      if (write(socket, &length_low, 1) != 1) {
         fprintf(stderr, "Error - EOF when sending token\n");
         return -1;
      };

      if (write(socket, &length_high, 1) != 1) {
         fprintf(stderr, "Error - EOF when sending token\n");
         return -1;
      };

      if (write_token(socket, &snd_token) != snd_token.length) {
         fprintf(stderr, "Error - EOF when sending token\n");
         return -1;
      };
   };

   if (maj_stat & GSS_S_CONTINUE_NEEDED) {
      if (read(socket, &length_low, 1) != 1) {
         fprintf(stderr, "Error - EOF when waiting for token\n");
         return -1;
      };
      if (read(socket, &length_high, 1) != 1) {
         fprintf(stderr, "Error - EOF when waiting for token\n");
         return -1;
      };
```

```
      length = length_low + length_high * 256;
      rcv_token.value = realloc(rcv_token.value, length);
      rcv_token.length = length;
      if (read_token(socket, &rcv_token) != rcv_token.length) {
         fprintf(stderr, "Error - EOF when waiting for token\n");
         return -1;
      };
   } else {
        done = 1;
   }
   };
};

/*****************/
/* function main */

int main(int argc, char * argv[]) {
   int sockfd;
   struct  sockaddr_inserv_addr;
   struct servent * server;
   struct hostvent * host;
   int i, c;
   int errflg = 0;
   int service_level;
   char * server_name = "scenario1_server_princ"; /* set server name */
   const char * clientname = "scenario1_princ";   /* set client name */
   const char * keytab =     "./scenario1_tab";   /* set keytab file */
   const char * host_param = "tivdce1";           /* set hostname */
   const char * port_param = "12345";             /* set port number */

/* get initial dce credential for client application */
if (login_to_dce(
   (unsigned_char_p_t)clientname,
   (unsigned_char_p_t)keytab)) {
      fprintf(stderr, "Error - login_to_dce() \n");
      return -1;
}

/* prepare socket creation */
for (i=0; isdigit(port_param[i]); i++);
   if (port_param[i]) {
      if ((server = getservbyname(port_param, "tcp")) == NULL) {
         fprintf(stderr, "Couldn't locate %s\n",port_param);
         return EXIT_FAILURE;
      };
      port = server->s_port;
      strncpy(port_name, server->s_name, sizeof(port_name));
   } else {
      port = 0;
```

```
      for (i=0; port_param[i]; i++) {
         port = port * 10 + value(port_param[i]);
      };
      if ((server = getservbyport(port, "tcp")) == NULL) {
         strncpy(port_name, "<unnamed service>", sizeof(port_name));
      } else {
         strncpy(port_name, server->s_name, sizeof(port_name));
      };
   };

   /* Now to translate host_param to numeric form */
   for (i=0; host_param[i] != '\0'; i++) {
      if (!isdigit(host_param[i]) && (host_param[i] != '.')) break;
   };

   host_entry = gethostbyname(host_param);
   memset((unsigned char *)&serv_addr, 0, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;
   memcpy((unsigned char *)&serv_addr.sin_addr.s_addr,
          (unsigned char *)host_entry->h_addr,4);
   serv_addr.sin_port = htons((short) port);

   if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   err_dump("client: can't open stream socket");

   if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
   err_sys("client: Can't connect to server ");

   /* initialize communication, by authenticating with server */
   if (define_service(
         sockfd,
         &security_context,
         server_name)) {
      exit (EXIT_FAILURE);
   };

   fprintf(stdout, "Connection open...\n");
   fflush(stdout);

   /* call str_cli to exchange protected application data with server */
   str_cli(stdin, sockfd, SERVICE_SECURE, security_context);
   close(sockfd);
   return EXIT_SUCCESS;
}
```

DCE dependent application server

Description: Server application with DCE dependencies
File name: server_s1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <dce/gssapi.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <sys/socket.h>
#include "utils_s1.h"

#define value(x) (((x>='0')&&(x<='9'))?x-'0':-1)

const char * port_param = "12345";                   /* set port number */
const char * keytab = "./scenario1_tab";             /* set keytab file */
const char * server_name = "scenario1_server_princ"; /* set server principal */

char port_name[512];
int  port;

/***************************/
/* function accept_service */

int accept_service(
   int socket,
   gss_ctx_id_t * sec_ctx,
   gss_cred_id_t cred_handle) {

   int                  done;
   unsigned char        length_low, length_high;
   unsigned             length;
   gss_buffer_desc      snd_token;
   gss_buffer_desc      rcv_token;
   gss_name_t           client;
   gss_buffer_desc      client_name;
   OM_uint32            maj_stat, min_stat;

   *sec_ctx = GSS_C_NO_CONTEXT;
   rcv_token.length = 0;
   rcv_token.value = NULL;
   snd_token.length = 0;
   snd_token.value = NULL;
   done = 0;

   while (!done) {
```

```
if (read(socket, &length_low, 1) != 1) {
   fprintf(stderr, "Error reading token length\n");
   return -1;
};

if (read(socket, &length_high, 1) != 1) {
   fprintf(stderr, "Error reading token length\n");
   return -1;
};

length = length_low + length_high * 256;
rcv_token.value = realloc(rcv_token.value, length);
rcv_token.length = length;

if (read_token(socket, &rcv_token) != rcv_token.length) {
   fprintf(stderr, "Error - EOF when waiting for token\n");
   return -1;
};

maj_stat = gss_accept_sec_context(
              &min_stat,
              sec_ctx,
              cred_handle,
              &rcv_token,
              GSS_C_NO_CHANNEL_BINDINGS,
              &client,
              NULL,
              &snd_token,
              NULL,
              NULL,
              NULL);
if (GSS_ERROR(maj_stat)) {
   display_error("gss_accept_sec_context", maj_stat, min_stat);
   return -1;
};

if (!(maj_stat & GSS_S_CONTINUE_NEEDED)) done = 1;

if (snd_token.length != 0) {
   length_low = snd_token.length & 0xff;
   length_high = snd_token.length >> 8;

   if (write(socket, &length_low, 1) != 1) {
      fprintf(stderr, "Error - EOF when writing token length\n");
      return -1;
   };

   if (write(socket, &length_high, 1) != 1) {
```

```
         fprintf(stderr, "Error - EOF when writing token length\n");
         return -1;
      };

      if (write_token(socket, &snd_token) != snd_token.length) {
         fprintf(stderr, "Error - EOF when sending token\n");
         return -1;
      };
   };

   }; /* end while (!done) */

   maj_stat = gss_display_name(
                 &min_stat,
                 client,
                 &client_name,
                 NULL);
   if (GSS_ERROR(maj_stat)) {
      display_error("gss_display_name", maj_stat, min_stat);
      return -1;
   };

   fprintf(
      stdout,
      "Accepted authenticated connection from %.*s\n",
      client_name.length,
      client_name.value);

} /* end accept_service */


/******************/
/* function main */

int main(int argc, char * argv[]) {
   int c,i,sockfd,newsockfd,clilen,childpid;
   int service_level = 2;
   int errflg = 0;
   struct sockaddr_in cli_addr, serv_addr;
   struct servent * server;
   gss_ctx_id_t security_context;

   OM_uint32 min_stat, maj_stat;
   gss_cred_id_t cred_handle = GSS_C_NO_CREDENTIAL;
   gss_name_t server_name_t;
   gss_buffer_desc name_buffer;

   extern int optind, opterr;
   extern char * optarg;
```

```
for (i=0; isdigit(port_param[i]); i++);
if (port_param[i]) {
   if ((server = getservbyname(port_param, "tcp")) == NULL) {
      fprintf(stderr, "Couldn't locate %s\n",port_param);
   exit (EXIT_FAILURE);
   };

   port = server->s_port;
   strncpy(port_name, server->s_name, sizeof(port_name));
} else {
   port = 0;
   for (i=0; port_param[i]; i++) {
      port = port * 10 + value(port_param[i]);
   };

   if ((server = getservbyport(port, "tcp")) == NULL) {
      strncpy(port_name, "<unnamed service>", sizeof(port_name));
   } else {
      strncpy(port_name, server->s_name, sizeof(port_name));
   };
  };


if (server_name != NULL) {

   name_buffer.length = strlen(server_name);
   name_buffer.value = (char *) server_name;

   maj_stat = gss_import_name(
                 &min_stat,
                 &name_buffer,
                 GSS_C_NULL_OID,
                 &server_name_t);
      if (GSS_ERROR(maj_stat)) {
        display_error("gss_import_name", maj_stat, min_stat);
        exit(EXIT_FAILURE);
      };


   if (keytab != NULL) {
      maj_stat = gssdce_register_acceptor_identity(
                 &min_stat,
                 server_name_t,
                 NULL,
                 (char *)keytab);
      if (GSS_ERROR(maj_stat)) {
         display_error("gssdce_register_identity", maj_stat, min_stat);
         exit (EXIT_FAILURE);
```

```
      };
   };

   maj_stat = gss_acquire_cred(
                  &min_stat,
                  server_name_t,
                  24*60*60,
                  GSS_C_NULL_OID_SET,
                  GSS_C_ACCEPT,
                  &cred_handle,
                  NULL,
                  NULL);
      if (GSS_ERROR(maj_stat)) {
         display_error("gss_acquire_cred", maj_stat, min_stat);
         exit (EXIT_FAILURE);
      };
};

fprintf(
   stdout,
   "Mirror server started on port %d, %s\n",
   port,
   port_name);

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   err_dump("server: can't open stream socket");

memset((unsigned char *)&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons((short) port);

if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
  err_dump("server: Can't bind local address");

listen(sockfd, 5);

for (;;) {
 clilen = sizeof(cli_addr);
 newsockfd =
 accept(sockfd, (struct sockaddr *)&cli_addr, (unsigned long *)&clilen);
 if (newsockfd < 0) err_dump("server: accept error");
 if ((childpid = fork()) < 0) {
   err_dump("server: fork error");
 } else if (childpid == 0) {
   close(sockfd);

/* authenticate client */
accept_service(
```

```
      newsockfd,
      &security_context,
      cred_handle);

      fprintf(stdout, "Accepted connection from host %d.%d.%d.%d\n",
         ((unsigned char *)&cli_addr.sin_addr.s_addr)[0],
         ((unsigned char *)&cli_addr.sin_addr.s_addr)[1],
         ((unsigned char *)&cli_addr.sin_addr.s_addr)[2],
         ((unsigned char *)&cli_addr.sin_addr.s_addr)[3]);

          /* call str_s1 to exchange protected application data with client */
         str_s1(newsockfd, service_level, security_context);
         exit(0);
      }
    close(newsockfd);
   }
} /* end main */
```

Authorization module with DCE dependencies

Description: Authorization module with DCE dependencies
File name: dce_authz.c

```
#include <stdio.h>
#include <string.h>
#include <dce/gssapi.h>
#include <dce/aclbase.h>
#include <dce/uuid.h>
#include <dce/aclif.h>
#include <dce/pgo.h>

/* The name of a sample object,used */
#define SAMPLE_GROUP_NAME    "scenario1_clients"

/* The permissions of the sample user that will be stored in a
   sample ACL */
#define OBJ_USER_PERMS sec_acl_perm_read | sec_acl_perm_write \
                | sec_acl_perm_delete | sec_acl_perm_test \
                | sec_acl_perm_execute \
                | sec_acl_perm_control

/*****************************************************************************/
/* ANSI-C style prototypes for functions private to this module...         */

void server_get_uuids(
   unsigned_char_t *,
   uuid_t *,
```

```
   uuid_t *,
   unsigned32 *);

void setup_sample_acl (
   sec_acl_t **,
   unsigned32 *);

/****************************************************************************/

/* The UUID of the sample ACL: */
uuid_t sample_acl_uuid = { /* d56f71da-a012-11ce-92fb-000000806635 */
   0xd56f71da,
   0xa012,
   0x11ce,
   0x92,
   0xfb,
   0x00,
   0x00,
   0x00,
   0x80,
   0x66,
   0x35};

/* The UUID of the sample ACL manager for the sample ACL:*/
uuid_t sample_acl_mgr_uuid = { /* 8091da9a-a012-11ce-9e9d-000000806635 */
   0x8091da9a,
   0xa012,
   0x11ce,
   0x9e,
   0x9d,
   0x00,
   0x00,
   0x00,
   0x80,
   0x66,
   0x35};



void setup_sample_acl(
   sec_acl_t  **sample_acl,    /* To return sample ACL */
   unsigned32 *status){

   uuid_t realm_uuid;  /* For the realm's UUID (from the registry) */
   uuid_t group_uuid;  /* For the group's UUID (from the registry) */
   sec_acl_entry_t sample_entry;
   *status = error_status_ok;

   /* Initialize sample ACL with one sample entry*/
```

```
   (*sample_acl) = (sec_acl_t *)malloc(sizeof(sec_acl_t));
   (*sample_acl)->sec_acl_entries =
      (sec_acl_entry_t *) malloc(sizeof(sec_acl_entry_t));
   (*sample_acl)->num_entries = 1;

   /* Get UUIDs that will be stored in the sample ACL and sample entry */
   server_get_uuids(
      (unsigned_char_t *) SAMPLE_GROUP_NAME,
      &group_uuid,
      &realm_uuid,
      status);
   if (*status != error_status_ok){
      fprintf(
         stderr,
         "Error - server_get_uuids(), status: %d\n",*status);
         return;
   }

   /* Fill values in the sample ACL */
   (*sample_acl)->default_realm.uuid = realm_uuid;
   (*sample_acl)->sec_acl_manager_type = sample_acl_mgr_uuid;

   /* Fill values in the sample entry */
   sample_entry.perms = OBJ_USER_PERMS;
   sample_entry.entry_info.entry_type = sec_acl_e_type_group;
   sample_entry.entry_info.tagged_union.id.uuid = group_uuid;
   sample_entry.entry_info.tagged_union.id.name = NULL;

   (*sample_acl)->sec_acl_entries[0] = sample_entry;
}

void server_get_uuids(
   unsigned_char_t *g_name,    /* Simple group name.*/
   uuid_t *g_id,               /* group UUID returned here.*/
   uuid_t *c_id,               /* cell (realm) UUID returned here */
   unsigned32 *status){
   char *cell_name;            /* For local cell name. */
   sec_rgy_handle_t rhandle;   /* For registry server handle. */

   /* Get the cell name */
   dce_cf_get_cell_name(
      &cell_name,
      status);
   if (*status != error_status_ok){
      fprintf(
         stderr,
         "Error - dce_cf_get_cell_name(), status: %d\n",*status);
         return;
   }
```

```
   /* Bind to the cell's registry */
   sec_rgy_site_open(cell_name, &rhandle, status);
   if (*status != error_status_ok){
      fprintf(
         stderr,
         "Error - sec_rgy_site_open(), status: %d\n",*status);
         return;
   }

   /* Get the cell (realm) UUID */
   sec_id_parse_name(
      rhandle,
      cell_name,
      NULL,
      c_id,
      NULL,
      NULL,
      status);

   /* Free the cell name */
   free(cell_name);

   /* Get the user UUID */
   sec_rgy_pgo_name_to_id(
      rhandle,
      sec_rgy_domain_group,
      g_name,
      g_id,
      status);

   if (*status != error_status_ok){
       fprintf(
          stderr,
          "Error - dsec_rgy_pgo_name_to_id(), status: %d\n",*status);
          return;
   }
}

/* service is_client_authorized, used by external moduls */

boolean32 is_client_authorized(
   gss_ctx_id_t        security_context,
   sec_acl_permset_t   desired_perms){

   rpc_authz_cred_handle_t           cred_h;
   sec_acl_t                         *sample_acl;
   rpc_authz_cred_handle_t           output_cred;
   OM_uint32                         maj_stat, min_stat;
```

```
   error_status_t                    status;
   sec_acl_permset_t                 perms  = dce_acl_c_no_permissions;

   /* Set up sample a ACL                        */
   setup_sample_acl(&sample_acl, &status);
   if (status != error_status_ok){
      fprintf(
         stderr,
         "Error - setup_sample_acl(), status: %d\n",status);
         return false;
   }

/* get PAC of caller */
   maj_stat  = gssdce_extract_creds_from_sec_context(
      &min_stat,
      security_context,
      &output_cred);
   if (GSS_ERROR(maj_stat)) {
      display_error(
         "gssdce_extract_creds_from_sec_context", maj_stat, min_stat);
          exit (EXIT_FAILURE);
    };

/* get permission of caller */
   dce_acl_inq_permset_for_creds(
      output_cred,
      sample_acl,
      NULL,
      NULL,
      sec_acl_posix_no_semantics,
      &perms,
      &status);

   if (status != error_status_ok){
       fprintf(
          stderr,
          "Error - dce_acl_inq_permset_for_creds(), status: %d\n",status);
          return false;
   }
   printf("granted perms: %x\n",perms);

   /* If desired permissions are included in caller permissions,
      caller is authorized */
   if ( (desired_perms & perms) == desired_perms ){
       printf("access granted.\n");
       return true;
   }
    return false;
}
```

Utility source of the DCE dependent application

Description: Utility module with DCE dependencies
File name: utils_s1.c

```
#include <stdio.h>
#include <dce/gssapi.h>
#include <dce/daclif.h>
#include "utils_s1.h"
#include <sys/types.h>
#ifdef WIN32
# include <winsock.h>
# undef write
# define write(fd, bufptr, buflen) send(fd, bufptr, buflen, 0)
# undef read
# define read(fd, bufptr, buflen) recv(fd, bufptr, buflen, 0)
# undef close
# define close(fd) closesocket(fd)
#else
# include <sys/socket.h>
# include <netinet/in.h>
# include <arpa/inet.h>
# include <netdb.h>
#endif
#ifndef ECHO_INET_H_
#define ECHO_INET_H_
#endif
#define DEFAULT_ECHO_PORT "12345"
#define SERVICE_INSECURE 1
#define SERVICE_SECURE 2

#ifndef WIN32
#define min(a,b) (((a)<(b))?(a):(b))
#endif

#include <dce/daclif.h>

/* defined in dce_authz */
extern boolean32  is_client_authorized(
   gss_ctx_id_t       security_context,
   sec_acl_permset_t  desired_perms);

/* function READN - Read nbytes from file descriptor FD */
int readn(
   register int fd,
   register char * ptr,
   register int nbytes) {

   int nleft, nread;
```

```
   nleft = nbytes;
   while (nleft > 0) {
      nread = read(fd, ptr, nleft);
      if (nread < 0) return nread;
      else if (nread == 0) break; /* EOF */
   nleft -= nread;
   ptr += nread;
   }
   return (nbytes - nleft);
}

/* function WRITEN - Write nbytes to file descriptor FD */
int writen(
   register int fd,
   register char * ptr,
   register int nbytes) {

   int nleft, nwritten;
   nleft = nbytes;
   while (nleft > 0) {
      nwritten = write(fd, ptr, nleft);
      if (nwritten <= 0) return nwritten;
      nleft -= nwritten;
      ptr += nwritten;
   }
   return (nbytes - nleft);
}

/* function READLINE - Read from  nbytes to file descriptor FD */
int readline(
   register int fd,
   register char * ptr,
   register int maxlen) {

   int n, rc;
   char    c;
   for (n=1; n<maxlen; n++) {
      if ((rc=read(fd, &c, 1)) == 1) {
         *ptr++ = c;
         if (c == '\n') break;
         } else if (rc == 0) {
           if (n == 1) return 0; /* EOF, no data read */
           else break;           /* EOF, some data was read */
       } else return -1;            /* error */
   }
   *ptr = 0;
   return n;
}
```

```
#define MAXLINE 512

/* function str_s1 - receive message on server side, decrypt message and */
/* authorize */

void str_s1(
   int sockfd,
   int service_level,
   gss_ctx_id_t security_context) {

   int n,i;
   char line[MAXLINE];
   char enil[MAXLINE]; /* string for mirror*/
   OM_uint32 maj_stat, min_stat;
   gss_buffer_desc token, message;
   unsigned char length_low, length_high;

   for (;;) {
      if (service_level == SERVICE_SECURE) {
      n = read(sockfd, &length_low,1);
         if (n == 0) return; /* Connection closed */
         if (n != 1) {
            fprintf(stderr, "Error reading token length\n");
            exit(EXIT_FAILURE);
         };
         if (read(sockfd, &length_high,1) != 1) {
            fprintf(stderr, "Error reading token length\n");
            exit(EXIT_FAILURE);
         };
         token.length = length_high * 256 + length_low;
         token.value = malloc(token.length);
         if (read_token(sockfd, &token) != token.length) {
            fprintf(stderr, "Error reading token\n");
            exit(EXIT_FAILURE);
         };
         maj_stat = gss_unseal(&min_stat,
            security_context,
            &token,
            &message,
            NULL,
            NULL);
         if (GSS_ERROR(maj_stat)) {
            display_error("gss_unseal", maj_stat, min_stat);
            exit (EXIT_FAILURE);
         };

         if (!is_client_authorized(security_context, sec_acl_perm_control)) {
            fprintf(stderr, "Error, Client not autorized\n");
            strncpy(enil, "Sorry, no bonus.\n\0", MAXLINE);
```

```
            n=strlen(enil);
         }else {
            n = min(MAXLINE, message.length);
            strncpy(line, message.value, MAXLINE);
            for (i=0; i<n; enil[i++]=line[n-i-2]); /*fill enil*/
            enil[n-1]='\n';
          }
          } else {
            n = readline(sockfd, line, MAXLINE);
      };
      if (n == 0) return; /* Connection terminated */
      else if (n < 0)
         err_dump("str_s1: readline error");
      if (writen(sockfd, enil, n) != n)
         err_dump("str_s1: writen error");
   }
}

/* function str_cli */
/* get input message from stdin, encrypt message, and send to server */

void str_cli(
   register FILE *fp,
   register int sockfd,
   int service_level,
   gss_ctx_id_t security_context) {

   int n;
   char sendline[MAXLINE];
   char recvline[MAXLINE+1];
   gss_buffer_desc token;
   gss_buffer_desc message;
   OM_uint32 maj_stat, min_stat;
   unsigned char length_low;
   unsigned char length_high;

   while (fgets(sendline, MAXLINE, fp) != NULL) {
   n = strlen(sendline);
   if (service_level == SERVICE_SECURE) {
      message.length = n;
      message.value = sendline;
      maj_stat = gss_seal(
         &min_stat,
         security_context,
         1,                  /* encryption turned on */
         GSS_C_QOP_DEFAULT,
         &message,
         NULL,
         &token);
```

```
      if (GSS_ERROR(maj_stat)) {
         display_error("gss_seal", maj_stat, min_stat);
         exit (EXIT_FAILURE);
      };
      length_low = token.length & 0xff;
      length_high = token.length >> 8;
     if (write(sockfd, &length_low, 1) != 1) {
        fprintf(stderr, "Error writing length\n");
        exit(EXIT_FAILURE);
      };
      if (write(sockfd, &length_high, 1) != 1) {
        fprintf(stderr, "Error writing length\n");
        exit(EXIT_FAILURE);
      };
      write_token(sockfd, &token);
      gss_release_buffer(&min_stat, &token);
    } else {
      if (writen(sockfd, sendline, n) != n)
        err_sys("str_cli: writen error on socket");
    }
    n = readline(sockfd, recvline, MAXLINE);
    if (n < 0) err_dump("str_cli: readline error");
    recvline[n] = 0;
    fputs(recvline, stdout);
    }
    if (ferror(fp)) err_sys("str_cli: error reading file");
}

int read_token(
   int socket,
   gss_buffer_t token) {
   return readn(socket, (char *)token->value, token->length);
}

int write_token(
   int socket,
   gss_buffer_t token) {
  return writen(socket, (char *)token->value, token->length);
}

void display_error(
   char * where,
   OM_uint32 maj,
   OM_uint32 min) {
   OM_uint32 maj_stat, min_stat;
   gss_buffer_desc status_buffer;
   int ctx = 0;
   fprintf(stderr, "Errors detected in %s\n", where);
   do {
```

```
      if (GSS_ERROR(gss_display_status(
         &min_stat,
         maj,
         GSS_C_GSS_CODE,
         GSS_C_NULL_OID,
         &ctx,
         &status_buffer))) {
         fprintf(stderr, "Error translating major status code (%X)\n", maj);
         ctx = 0;
      } else {
         fprintf(stderr, "%.*s\n", status_buffer.length, status_buffer.value);
         gss_release_buffer(&min_stat, &status_buffer);
   };
   } while (ctx != 0);
   if (GSS_ERROR(gss_display_status(
      &min_stat,
      min,
      GSS_C_MECH_CODE,
      GSS_C_NULL_OID,
      &ctx,
      &status_buffer))) {
         fprintf(stderr, "Error translating minor status code (%X)\n", min);
   } else {
      fprintf(stderr, "%.*s\n", status_buffer.length, status_buffer.value);
      gss_release_buffer(&min_stat, &status_buffer);
   };
}
```

Header file for utility source

Description: Utility module with DCE dependencies
File name: utils_s1.h

```
#ifdef WIN32
#include <dce/rpcexc.h>
#endif
#ifndef _ECHO_UTILS_
#define _ECHO_UTILS_
#include <dce/gssapi.h>
#include <stdio.h>

#define err_sys(x) {perror(x); exit (1);}
#define err_dump(x) {fprintf(stderr, x); exit (1);}

int readn(register int fd,
   register char * ptr,
   register int nbytes);
```

```
int writen(register int fd,
   register char * ptr,
   register int nbytes);

int readline(register int fd,
   register char * ptr,
   register int maxlen);

void str_s1(int sockfd, int service_level, gss_ctx_id_t security_context);

void str_cli(register FILE *fp,
   register int sockfd,
   int service_level,
   gss_ctx_id_t security_context);

int read_token(int socket, gss_buffer_t tok);

int write_token(int socket, gss_buffer_t tok);

void display_error(char * where, OM_uint32 maj, OM_uint32 min);
#endif
```

Revised application without DCE dependencies

The files are:

- s1.mak contains information about how the application is compiled and linked on the Windows operating system.
- Makefile contains information about how the application is compiled and linked on the AIX operating system.
- client_s1.c contains the client code. After acquiring initial credentials and creating a shared security context with the server, the application client exchanges messages with the application server via socket communication.
- server_s1.c contains the server code. The application server acquires initial credentials and accepts a shared security context with the client. Then it listens for application messages from the client. Furthermore, it uses the azn_authz.c module to authorize the client in case of a request.
- azn_authz.c contains the code for the authorization module. Upon a client request, the client principal's credentials are looked up and compared with the desired permissions. Additionally, initialization and shutdown services for aznAPI are provided.
- utils_s1.c provides common functions for socket communication, and encryption and decryption are handled in this module.

- utils_s1.h represents the header file for utils_s1.c.
- remote.conf configuration file tells the application how to communicate with IBM Tivoli Access Manager.

Makefile for application client

Description: Makefile for Windows application client
File name: s1.mak

```
!include <NTWIN32.MAK>

!if "$(CPU)" == "ALPHA"
TARGET=alpha
SWITCHES=-DALPHA
!else
TARGET=intel
SWITCHES=-D_X86=1 -DM_I86
!endif

_LIBS_= \
   wsock32.lib \
   $(conlibsdll) secur32.lib


CFLAGS= -I. $(cflags:-W3=) $(cvarsdll) $(SWITCHES) -Zi -Od -Gz -nologo

LINK = link $(ldebug) $(conlflags)

all: client_s1.exe

client  : client_s1.exe

client_s1.exe : client_s1.obj utils_s1.obj
   $(LINK) /out:$@ $** $(_LIBS_)

client_s1.obj: client_s1.c
   $(CC) $(CFLAGS) client_s1.c

utils_s1.obj: utils_s1.c
   $(CC) $(CFLAGS) utils_s1.c

clean :
   del *.obj

clobber :
   del *.obj
   del *.exe
```

Makefile for application server

Description: Makefile for AIX application server
File name: Makefile

```
CC=/usr/vacpp/bin/xlC_r4

CFLAGS= -I. -g -o

all : server

server : server_s1.o utils_s1.o azn_authz.o
   $(CC) ${CFLAGS} server_s1 server_s1.o utils_s1.o azn_authz.o -lgssapi_krb5
-lpdauthzn

server_s1.o :
   $(CC) -c  server_s1.c

utils_s1.o :
   $(CC) -c utils_s1.c

azn_authz.o :
   $(CC) -I/usr/include/PolicyDirector -c azn_authz.c

clean :
   rm -f *.o

clobber :
   rm -f server_s1 *.o
```

Revised application client

Description: Revised client program without DCE dependencies
File name: client_s1.c

```
#include "utils_s1.h"

#define value(x) (((x>='0') && (x<='9'))?x-'0':-1)
#define MAX_HOST_LENGTH 64

char *         port_param = NULL;
char *         host_param = NULL;
int            port;
char           port_name[512];
unsigned       host_addr;
struct         hostent * host_entry = NULL;
```

```
/*##########################################################################*
/
/* function define_service */
/* exchanges authentication token with server for mutual authentication */

int define_service(
   int s,
   const char * server_name,
   CtxtHandle *gss_context) {

   SecBuffer               send_tok, recv_tok;
   SecBufferDesc           input_desc, output_desc;
   OM_uint32               maj_stat;
   CredHandle              cred_handle;
   TimeStamp               expiry;
   OM_uint32               ret_flags;
   PCtxtHandle             context_handle = NULL;

   OM_uint32 deleg_flag = ( ISC_REQ_MUTUAL_AUTH |
                           ISC_REQ_ALLOCATE_MEMORY |
                           ISC_REQ_CONFIDENTIALITY |
                           ISC_REQ_REPLAY_DETECT);

   input_desc.cBuffers = 1;
   input_desc.pBuffers = &recv_tok;
   input_desc.ulVersion = SECBUFFER_VERSION;

   recv_tok.BufferType = SECBUFFER_TOKEN;
   recv_tok.cbBuffer = 0;
   recv_tok.pvBuffer = NULL;

   output_desc.cBuffers = 1;
   output_desc.pBuffers = &send_tok;
   output_desc.ulVersion = SECBUFFER_VERSION;

   send_tok.BufferType = SECBUFFER_TOKEN;
   send_tok.cbBuffer = 0;
   send_tok.pvBuffer = NULL;

   cred_handle.dwLower = 0;
   cred_handle.dwUpper = 0;


   maj_stat = AcquireCredentialsHandle(
      NULL,                      // no principal name
      "Kerberos",                // package name
      SECPKG_CRED_OUTBOUND,
      NULL,                      // no logon id
      NULL,                      // no auth data
```

```
   NULL,                       // no get key fn
   NULL,                       // noget key arg
   &cred_handle,
   &expiry);
if (maj_stat != SEC_E_OK) {
   fprintf(stderr,"error acquiring credentials",maj_stat);
   return (-1);
}

/*
 * Perform the context-establishement loop.
 */
gss_context->dwLower = 0;
gss_context->dwUpper = 0;

do
{
   maj_stat = InitializeSecurityContext(
        &cred_handle,
        context_handle,
        (char *)server_name,
        deleg_flag,
        0,          // reserved
        SECURITY_NATIVE_DREP,
        &input_desc,
        0,          // reserved
        gss_context,
        &output_desc,
        &ret_flags,
        &expiry);

   if (recv_tok.pvBuffer) {
      free(recv_tok.pvBuffer);
      recv_tok.pvBuffer = NULL;
      recv_tok.cbBuffer = 0;

   }

   context_handle = gss_context;

   if (maj_stat!=SEC_E_OK && maj_stat!=SEC_I_CONTINUE_NEEDED) {
      fprintf(stderr, "error initializing context %d", maj_stat);
      FreeCredentialsHandle(&cred_handle);
      return -1;
   }

   if (send_tok.cbBuffer != 0) {
      printf("Sending init_sec_context token (size=%d)...",
             send_tok.cbBuffer);
```

```
            if (send_token(s, &send_tok) < 0) {
               FreeContextBuffer(send_tok.pvBuffer);
               FreeCredentialsHandle(&cred_handle);
               return -1;
            }
         }

         FreeContextBuffer(send_tok.pvBuffer);
         send_tok.pvBuffer = NULL;
         send_tok.cbBuffer = 0;

         if (maj_stat == SEC_I_CONTINUE_NEEDED) {
            printf("continue needed...");
            if (recv_token(s, &recv_tok) < 0) {
               printf("last token not received...");
               FreeCredentialsHandle(&cred_handle);
               return -1;
            }
         }
         printf("\n");

   } while (maj_stat == SEC_I_CONTINUE_NEEDED);

   FreeCredentialsHandle(&cred_handle);
   return 0;
};


/*##############################################################################*
/
/* function main */

int main(int argc, char * argv[]) {
   int                    sockfd;
   struct sockaddr_in     serv_addr;
   struct servent *       server;
   //struct hostvent *      host;
   int                    i;
   int                    errflg = 0;
   CtxtHandle             security_context;
   WSADATA                socket_data;
   USHORT                 version_required = 0x0101;
   int                    err;
   const char * server_name = "scenario1_server_princ"; /* set server name */
   const char * host_param = "tivdce3";                /* set hostname */
   const char * port_param = "12345";                  /* set port number */


/* The WSAStartup function initiates use of WS2_32.DLL by a process.*/
```

```
err = WSAStartup(version_required, &socket_data);
if (err) {
   fprintf(stderr,"Failed to initailize WSA: %d\n",err);
   return (-1);
}

/* prepare socket creation */

for (i=0; isdigit(port_param[i]); i++);
   if (port_param[i]) {
      if ((server = getservbyname(port_param, "tcp")) == NULL) {
         fprintf(stderr, "Couldn't locate %s\n",port_param);
         return EXIT_FAILURE;
      };
      port = server->s_port;
      strncpy(port_name, server->s_name, sizeof(port_name));
   } else {
      port = 0;
      for (i=0; port_param[i]; i++) {
         port = port * 10 + value(port_param[i]);
      };
      if ((server = getservbyport(port, "tcp")) == NULL) {
         strncpy(port_name, "<unnamed service>", sizeof(port_name));
      } else {
         strncpy(port_name, server->s_name, sizeof(port_name));
      };
   };


   /* Now to translate host_param to numeric form */
   for (i=0; host_param[i] != '\0'; i++) {
      if (!isdigit(host_param[i]) && (host_param[i] != '.')) break;
   };

   host_entry = gethostbyname(host_param);
   memset((unsigned char *)&serv_addr, 0, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;

   memcpy((unsigned char *)&serv_addr.sin_addr.s_addr,
          (unsigned char *)host_entry->h_addr,4);
   serv_addr.sin_port = htons((short) port);


   if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   err_dump("client: can't open stream socket");


   if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
   err_sys("client: Can't connect to server ");
```

```
   /* initialize secure communication, by authenticating with server*/
    if (define_service(
         sockfd,
         server_name,
         &security_context)) {
      exit (EXIT_FAILURE);
   };

   fprintf(stdout, "Connection open...\n");
   fflush(stdout);

   /* start exchanging protected application data with server */
   str_cli(stdin,sockfd,&security_context);

   close(sockfd);
   return EXIT_SUCCESS;
}
```

Revised application server

Description: Revised server program without DCE dependencies
File name: server_s1.c

```
#include "utils_s1.h"

#define value(x) (((x>='0')&&(x<='9'))?x-'0':-1)

const char * port_param = "12345";                   /* set port number */
const char * server_name = "scenario1_server_princ"; /* set server principal */

char port_name[512];
int  port;

void init_authz(char *);

/****************************/
/* function accept_service */

int accept_service(
   int socket,
   gss_ctx_id_t * sec_ctx,
   gss_cred_id_t cred_handle) {

   int                    done;
   unsigned char          length_low, length_high;
   unsigned               length;
   gss_buffer_desc        snd_token;
```

```
gss_buffer_desc      rcv_token;
gss_name_t           client;
gss_buffer_desc      client_name;
OM_uint32            maj_stat, min_stat;

*sec_ctx = GSS_C_NO_CONTEXT;
rcv_token.length = 0;
rcv_token.value = NULL;
snd_token.length = 0;
snd_token.value = NULL;
done = 0;

while (!done){

if (read(socket, &length_low, 1) != 1) {
   fprintf(stderr, "Error reading token length\n");
   return -1;
};


if (read(socket, &length_high, 1) != 1) {
   fprintf(stderr, "Error reading token length\n");
   return -1;
};

length = length_low + length_high * 256;
rcv_token.value = realloc(rcv_token.value, length);
rcv_token.length = length;

if (read_token(socket, &rcv_token) != rcv_token.length) {
   fprintf(stderr, "Error - EOF when waiting for token\n");
   return -1;
};

maj_stat = gss_accept_sec_context(&min_stat,
              sec_ctx,
              cred_handle,
              &rcv_token,
              GSS_C_NO_CHANNEL_BINDINGS,
              &client,
              NULL,
              &snd_token,
              NULL,
              NULL,
              NULL);
if (GSS_ERROR(maj_stat)) {
   display_error("gss_accept_sec_context", maj_stat, min_stat);
   return -1;
};
```

```
   if (!(maj_stat & GSS_S_CONTINUE_NEEDED)) done = 1;

   if (snd_token.length != 0) {
      length_low = snd_token.length & 0xff;
      length_high = snd_token.length >> 8;

      if (write(socket, &length_low, 1) != 1) {
         fprintf(stderr, "Error - EOF when writing token length\n");
         return -1;
      };

      if (write(socket, &length_high, 1) != 1) {
         fprintf(stderr, "Error - EOF when writing token length\n");
         return -1;
      };

      if (write_token(socket, &snd_token) != snd_token.length) {
         fprintf(stderr, "Error - EOF when sending token\n");
         return -1;
      };
   };

   }; /* end while (!done) */

   maj_stat = gss_display_name(
                 &min_stat,
                 client,
                 &client_name,
                 NULL);
   if (GSS_ERROR(maj_stat)) {
      display_error("gss_display_name", maj_stat, min_stat);
      return -1;
   };

   fprintf(
      stdout,
      "Accepted authenticated connection from %.*s\n",
      client_name.length,
      client_name.value);

} /* end accept_service */

/*****************/
/* function main */

int main(int argc, char * argv[]) {
   int            c,i,sockfd,newsockfd,clilen,childpid;
   int            errflg = 0;
```

```
struct         sockaddr_in cli_addr, serv_addr;
struct         servent * server;
gss_ctx_id_t   security_context;

OM_uint32 min_stat, maj_stat;
gss_cred_id_t cred_handle = GSS_C_NO_CREDENTIAL;
gss_name_t server_name_t;
gss_buffer_desc name_buffer;

extern int optind, opterr;
extern char * optarg;

init_authz("scenario1_princ"); /* Initialize authorization module */

for (i=0; isdigit(port_param[i]); i++);
if (port_param[i]) {
   if ((server = getservbyname(port_param, "tcp")) == NULL) {
      fprintf(stderr, "Couldn't locate %s\n",port_param);
      exit (EXIT_FAILURE);
   };

   port = server->s_port;
   strncpy(port_name, server->s_name, sizeof(port_name));
} else {
   port = 0;
   for (i=0; port_param[i]; i++) {
      port = port * 10 + value(port_param[i]);
   };

   if ((server = getservbyport(port, "tcp")) == NULL) {
      strncpy(port_name, "<unnamed service>", sizeof(port_name));
   } else {
      strncpy(port_name, server->s_name, sizeof(port_name));
   };
};


if (server_name != NULL) {

   name_buffer.length = strlen(server_name);
   name_buffer.value = (char *) server_name;

   maj_stat = gss_import_name(
                 &min_stat,
                 &name_buffer,
                 GSS_C_NULL_OID,
                 &server_name_t);
      if (GSS_ERROR(maj_stat)) {
         display_error("gss_import_name", maj_stat, min_stat);
```

```
         exit(EXIT_FAILURE);
      };

   maj_stat = gss_acquire_cred(
                 &min_stat,
                 server_name_t,
                 24*60*60,
                 GSS_C_NULL_OID_SET,
                 GSS_C_ACCEPT,
                 &cred_handle,
                 NULL,
                 NULL);
      if (GSS_ERROR(maj_stat)) {
         display_error("gss_acquire_cred", maj_stat, min_stat);
         exit (EXIT_FAILURE);
      };
};

fprintf(
   stdout,
   "Mirror server started on port %d, %s\n",
   port,
   port_name);

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
   err_dump("server: can't open stream socket");

memset((unsigned char *)&serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons((short) port);

if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
   err_dump("server: Can't bind local address");

listen(sockfd, 5);

for (;;) {
   clilen = sizeof(cli_addr);
   newsockfd =
   accept(sockfd, (struct sockaddr *)&cli_addr, (unsigned long *)&clilen);
 if (newsockfd < 0) err_dump("server: accept error");
 if ((childpid = fork()) < 0) {
    err_dump("server: fork error");
 } else if (childpid == 0) {
    close(sockfd);

/* authenticate with client */
accept_service(
```

```
      newsockfd,
      &security_context,
      cred_handle);

      fprintf(stdout, "Accepted connection from host %d.%d.%d.%d\n",
         ((unsigned char *)&cli_addr.sin_addr.s_addr)[0],
         ((unsigned char *)&cli_addr.sin_addr.s_addr)[1],
         ((unsigned char *)&cli_addr.sin_addr.s_addr)[2],
         ((unsigned char *)&cli_addr.sin_addr.s_addr)[3]);

          /* call str_s1 to exchange protected application data with client */
         str_s1(newsockfd, security_context);
         exit(0);
      }
    close(newsockfd);
   }

} /* end main */
```

Authorization module using aznAPI

Description: Revised authorization module without DCE dependencies
File name: azn_authz.c

```
#include <stdlib.h>
#include <stdio.h>
#include <strings.h>
#include <ctype.h>
#include <ogauthzn.h>
#include <aznutils.h>
#include <pdbaclmsg.h>

#ifndef FALSE
#define FALSE 0
#endif
#ifndef TRUE
#define TRUE 1
#endif

#define LINELEN 128

char prin_name[LINELEN];

/* check status and print a message. */
void
check_status(
   char *info,
   unsigned status){
```

```
   azn_string_t errstr = NULL;
   unsigned majerr, minerr;
   azn_status_t rc;

   if (status == AZN_S_COMPLETE)
      return;

   majerr = azn_error_major(status);
   minerr = azn_error_minor(status);

   printf("\n%s: ", info);

   rc = azn_error_get_string(status, &errstr);
   if (rc == AZN_S_COMPLETE && errstr) {
   printf("%s (0x%08x/0x%08x)\n\n", errstr, majerr, minerr);
   azn_release_string(&errstr);
   }
   else
      printf("API failure (0x%08x/0x%08x)\n\n", majerr, minerr);
}

/* Initialization of the znAPI */
void init_authz(char * principal_name){

   azn_status_t                status;
   int                         permitted;
   int                         i,x;
   azn_attrlist_h_t            initdata = AZN_C_INVALID_HANDLE;
   azn_attrlist_h_t            initinfo = AZN_C_INVALID_HANDLE;
   azn_string_t                *attrnames;

   printf("** Using remote replica (ivacld) **\n");

   for(i=1;i<sizeof(prin_name);i++)
      prin_name[i] = '\0';

   strcpy(prin_name,principal_name);

   /* Create handles to attribute lists */
   azn_attrlist_create(&initdata);

   /* Don't use a local replica, use the authorization server */

   status = azn_attrlist_add_entry(
      initdata,
      azn_init_mode,
      "remote");
   check_status("add attribute azn_init_mode", status);
   if (status != AZN_S_COMPLETE){
```

```
    exit(status);
  }

status = azn_attrlist_add_entry(
    initdata,
    azn_init_cfg_file,
    "./remote.conf");
check_status("add attribute azn_init_cfg_file", status);
if (status != AZN_S_COMPLETE)
    exit(status);

/* view the attribute list before initializing */
status = azn_attrlist_get_names(initdata, &attrnames);
if ( status != AZN_S_COMPLETE ){
check_status("azn_attrlist_get_names", status);
    exit (status);
}

printf("\n[[ Init attribute list ]]\n");
for (x=0; attrnames[x] != NULL; x++)
    printf("    [%02d]: %s\n", x, attrnames[x]);
    printf("\n");

azn_release_strings(&attrnames);

/* Start the service */
status = azn_initialize(initdata, &initinfo);
check_status("azn_initialize", status);
if (status != AZN_S_COMPLETE){
    fprintf(
       stderr,
       "Could not initialise the authorization service!\n", status);
       exit(1);
}

/* View the returned initinfo attribute list including
 * the AZN API client lib version. */

if (initinfo != AZN_C_INVALID_HANDLE){
status = azn_attrlist_get_names(initinfo, &attrnames);
if ( status == AZN_S_COMPLETE ) {
    printf("\n[[ Attribute list returned from azn_initialize ]]\n");
    for (x=0; attrnames[x] != NULL; x++)
    printf("    [%02d]: %s\n", x, attrnames[x]);
    printf("\n");
    azn_release_strings(&attrnames);
} else
      check_status("azn_attrlist_get_names", status);
   }
```

```
   /* delete handles to attribute lists */
   azn_attrlist_delete(&initdata);
   azn_attrlist_delete(&initinfo);
} /* end init_autz */

/* service is_client_authorized, used by external moduls */
int is_client_authorized(
   char * permission){

   azn_status_t              status;
   int                       permitted;
   char                      operation[LINELEN];
   char                      obj_name[LINELEN];
   azn_authdefault_t         default_minfo;
   azn_creds_h_t             creds = AZN_C_INVALID_HANDLE;

   azn_string_t              mech = NULL;
   azn_buffer_desc           buf = { 0, 0};

   /* terminate the input strings */
   operation[0] = (char)0;
   obj_name[0] = (char)0;

   /* Only the control flag on the ACL is checked */
   /* harcoded here, to keep the ptogram simple, and */
   /* other moduls independent from authz includes */
   strcpy(operation, azn_operation_control);


   /* protected AM resource, used for scenario1 */
   strcpy(obj_name, "/Scenario1/Resource1");

   /* default creds mechanism info */
   default_minfo.auth_method = "default_auth_method";
   default_minfo.authnmech_info = "default_authnmech_info";
   default_minfo.qop = "default_qop";
   default_minfo.user_info = "default_user_info";
   default_minfo.browser_info = "default_browser_info";
   default_minfo.ipaddr = 0x0a000002;

   /* Set mechanism ID to NULL so the aznAPI will determine
    * which registry (LDAP or URAF) that Policy Director is
    * configured to use. */
   mech = (char *)NULL;
   default_minfo.principal = prin_name;
   buf.length = sizeof(default_minfo);
   buf.value = (unsigned char *)&default_minfo;
```

```
/* create a credential */
status = azn_creds_create(&creds);
check_status("azn_creds_create", status);
if (status != AZN_S_COMPLETE) {
   fprintf(stderr, "Could not create a cred!\n");
   exit(1);
}

/* get a credential */
status = azn_id_get_creds(NULL, mech, &buf, &creds);
check_status("azn_id_get_creds", status);
if (status != AZN_S_COMPLETE) {
   fprintf(stderr, "Could not get creds!\n\n");
}

/*  "\nPlease build an action string from the following.\n"
     "       [a] azn_operation_attach\n"
     "       [b] azn_operation_browse\n"
     "       [B] azn_operation_bypasstod\n"
     "       [c] azn_operation_control\n"
     "       [T] azn_operation_traverse\n"
     "       [g] azn_operation_delegation\n"
     "       [v] azn_operation_view\n"
     "       [m] azn_operation_modify\n"
     "       [d] azn_operation_delete\n"
     "       [s] azn_operation_server_admin\n"
     "       [r] azn_operation_read\n"
     "       [x] azn_operation_execute\n"
     "       [l] azn_operation_list_directory\n"
     "       [W] azn_operation_password\n"
     "       [N] azn_operation_create\n"
     "       [A] azn_operation_add\n\n"  */

/* Perform standard authorization check */
status = azn_decision_access_allowed(
   creds,
   obj_name,
   operation,
   &permitted);
check_status("azn_decision_access_allowed", status);

status = azn_creds_delete(&creds);

if (permitted == AZN_C_PERMITTED) {
   printf("Permitted.\n\n");
   return TRUE;
}
else {
   printf("Not permitted.\n\n");
```

```
      return FALSE;
   }
} /* end is_client_authorized */

/* shutdown the aznAPI service */
void shutdown_authz(){
   azn_status_t                status;
   status = azn_shutdown();
   check_status("azn_shutdown", status);

}
```

Utility source of the revised application

Description: Utility module without DCE dependencies
File name: utils_s1.c

```
#
/* Common client and server utils. client code intended for use on windows only
*/

#include "utils_s1.h"

#define MAXLINE 512

/* non DCE function READN - Read nbytes from file descriptor FD */

int readn(register int fd,
      register char * ptr,
      register int nbytes) {
    int nleft, nread;

    nleft = nbytes;
    while (nleft > 0) {
    nread = read(fd, ptr, nleft);
    if (nread < 0) return nread;
    else if (nread == 0) break; /* EOF */

    nleft -= nread;
    ptr += nread;
    }
    return (nbytes - nleft);

}


/* non DCE function WRITEN - Write nbytes to file descriptor FD */
```

```
int writen(register int fd,
       register char * ptr,
       register int nbytes) {
    int nleft, nwritten;

    nleft = nbytes;

    while (nleft > 0) {
    nwritten = write(fd, ptr, nleft);
    if (nwritten <= 0) return nwritten;

    nleft -= nwritten;
    ptr += nwritten;
    }
    return (nbytes - nleft);

}


/* non DCE function READLINE - Read from  nbytes to file descriptor FD */

int readline(register int fd,
         register char * ptr,
         register int maxlen) {
    int n, rc;
    char    c;

    for (n=1; n<maxlen; n++) {
    if ((rc=read(fd, &c, 1)) == 1) {
        *ptr++ = c;
        if (c == '\n') break;
    } else if (rc == 0) {
        if (n == 1) return 0;   /* EOF, no data read            */
        else break;             /* EOF, some data was read      */
    } else return -1;           /* error                        */
    }
    *ptr = 0;
    return n;
}

#ifndef WIN32
/* function str_s1 */
/* receive message on server side, decrypt message, and authorize */

void str_s1(int sockfd, gss_ctx_id_t security_context) {
   int                     n,i;
   char                    line[MAXLINE];
   char                    enil[MAXLINE]; /* string for mirror*/
   OM_uint32               maj_stat, min_stat;
```

```
   gss_buffer_desc         token,  message;
   unsigned char           length_low, length_high;

   for (;;) {
       n = read(sockfd, &length_low,1);
       if (n == 0) return; /* Connection closed */
       if (n != 1) {
          fprintf(stderr, "Error reading token length\n");
          exit(EXIT_FAILURE);
       };
       if (read(sockfd, &length_high,1) != 1) {
          fprintf(stderr, "Error reading token length\n");
          exit(EXIT_FAILURE);
       };
       token.length = length_high * 256 + length_low;
       token.value = malloc(token.length);
       if (read_token(sockfd, &token) != token.length) {
          fprintf(stderr, "Error reading token\n");
          exit(EXIT_FAILURE);
       };
       maj_stat = gss_unseal(&min_stat,
               security_context,
               &token,
               &message,
               NULL,
               NULL);
       if (GSS_ERROR(maj_stat)) {
          display_error("gss_unseal", maj_stat, min_stat);
          exit (EXIT_FAILURE);
       };
       if (!is_client_authorized("azn_operation_control")) {
          fprintf(stderr, "Error, Client not autorized\n");
          strncpy(enil, "Sorry, no bonus.\n\0", MAXLINE);
          n=strlen(enil);
       }else {
          n = min(MAXLINE, message.length);
          strncpy(line, message.value, MAXLINE);
          for (i=0; i<n; enil[i++]=line[n-i-2]); /*fill enil*/
          enil[n-1]='\n';
        }
       if (n == 0) return; /* Connection terminated             */
       else if (n < 0)
          err_dump("str_s1: readline error");
       if (writen(sockfd, enil, n) != n)
          err_dump("str_s1: writen error");
   }
}
```

```
void display_error(char * where, OM_uint32 maj, OM_uint32 min) {
  OM_uint32 maj_stat, min_stat;
  gss_buffer_desc status_buffer;
  int ctx = 0;

  fprintf(stderr, "Errors detected in %s\n", where);
  do {

    if (GSS_ERROR(gss_display_status(&min_stat,
                    maj,
                    GSS_C_GSS_CODE,
                    GSS_C_NULL_OID,
                    (unsigned int *) &ctx,
                    &status_buffer))) {
      fprintf(stderr, "Error translating major status code (%X)\n", maj);
      ctx = 0;
    } else {
      fprintf(stderr, "%.*s\n", status_buffer.length, status_buffer.value);
      gss_release_buffer(&min_stat, &status_buffer);
    };
  } while (ctx != 0);

  if (GSS_ERROR(gss_display_status(&min_stat,
                  min,
                  GSS_C_MECH_CODE,
                  GSS_C_NULL_OID,
                  (unsigned int *) &ctx,
                  &status_buffer))) {
    fprintf(stderr, "Error translating minor status code (%X)\n", min);
  } else {
    fprintf(stderr, "%.*s\n", status_buffer.length, status_buffer.value);
    gss_release_buffer(&min_stat, &status_buffer);
  };

}

int read_token(int socket, gss_buffer_t token) {
  return readn(socket, (char *)token->value, token->length);
}

int write_token(int socket, gss_buffer_t token) {
  return writen(socket, (char *)token->value, token->length);
}


#else  /* WIN32, always a bit different */
/* function str_cli */
/* get input message, encrypt message and send to server */
```

```
void str_cli(
    register FILE *fp,
    register int sockfd,
    CtxtHandle * security_context) {

    int                     n;
    char                    sendline[MAXLINE];
    char                    recvline[MAXLINE+1];
    OM_uint32               maj_stat;
    SecBuffer               in_buf, out_buf;
    SecBuffer               wrap_bufs[3];
    SecBufferDesc           in_buf_desc;
    SecPkgContext_Sizes     sizes;
    gss_qop_t               qop_state;

   maj_stat = QueryContextAttributes(
       security_context,
       SECPKG_ATTR_SIZES,
       &sizes
       );
   if (maj_stat != SEC_E_OK) {
      fprintf(stderr, "error querying context attributes", maj_stat);
      exit (-1);
   }

   while (fgets(sendline, MAXLINE, stdin) != NULL) {
      n = strlen(sendline);

      in_buf.pvBuffer = sendline;
      in_buf.cbBuffer = n;


      //
      // Prepare to encrypt the message
      //
      in_buf_desc.cBuffers = 3;
      in_buf_desc.pBuffers = wrap_bufs;
      in_buf_desc.ulVersion = SECBUFFER_VERSION;

      wrap_bufs[0].cbBuffer = sizes.cbSecurityTrailer;
      wrap_bufs[0].BufferType = SECBUFFER_TOKEN;
      wrap_bufs[0].pvBuffer = malloc(sizes.cbSecurityTrailer);

      if (wrap_bufs[0].pvBuffer == NULL) {
         fprintf(stderr,"Failed to allocate space for security trailer\n");
         exit (-1);
      }

      wrap_bufs[1].BufferType = SECBUFFER_DATA;
```

```
wrap_bufs[1].cbBuffer = in_buf.cbBuffer;
wrap_bufs[1].pvBuffer = malloc(wrap_bufs[1].cbBuffer);

if (wrap_bufs[1].pvBuffer == NULL) {
   fprintf(stderr,"Couldn't allocate space for wrap message\n");
   exit (-1);
}

memcpy(
   wrap_bufs[1].pvBuffer,
   in_buf.pvBuffer,
   in_buf.cbBuffer);

wrap_bufs[2].BufferType = SECBUFFER_PADDING;
wrap_bufs[2].cbBuffer = sizes.cbBlockSize;
wrap_bufs[2].pvBuffer = malloc(wrap_bufs[2].cbBuffer);

if (wrap_bufs[2].pvBuffer == NULL) {
   fprintf(stderr,"Couldn't allocate space for wrap message\n");
   exit (-1);
}

maj_stat = EncryptMessage(
                   security_context,
                   0,
                   &in_buf_desc,
                   0);


if (maj_stat != SEC_E_OK) {
   fprintf(stderr,"Couldn't encrypt message %d\n", maj_stat);
   (void) closesocket(sockfd);
   (void) DeleteSecurityContext(security_context);
   exit (-1);
}

//
// Create the message to send to server
//

out_buf.cbBuffer =
wrap_bufs[0].cbBuffer +
wrap_bufs[1].cbBuffer +
wrap_bufs[2].cbBuffer;
out_buf.pvBuffer = malloc(out_buf.cbBuffer);

if (out_buf.pvBuffer == NULL) {
   fprintf(stderr,"Failed to allocate space for wrapepd message\n");
   exit (-1);
```

```
      }

      memcpy(
         out_buf.pvBuffer,
         wrap_bufs[0].pvBuffer,
         wrap_bufs[0].cbBuffer);
      memcpy(
         (PUCHAR) out_buf.pvBuffer + (int) wrap_bufs[0].cbBuffer,
         wrap_bufs[1].pvBuffer,
         wrap_bufs[1].cbBuffer);
      memcpy(
         (PUCHAR) out_buf.pvBuffer +
               wrap_bufs[0].cbBuffer +
               wrap_bufs[1].cbBuffer,
         wrap_bufs[2].pvBuffer,
         wrap_bufs[2].cbBuffer);

      /* Send to server */
      if (send_token(sockfd, &out_buf) < 0)
      {
         (void) closesocket(sockfd);
         (void) DeleteSecurityContext(security_context);
         exit (-1);
      }

      free(out_buf.pvBuffer);
      out_buf.pvBuffer = NULL;
      out_buf.cbBuffer = 0;
      free(wrap_bufs[0].pvBuffer);
      wrap_bufs[0].pvBuffer = NULL;
      free(wrap_bufs[1].pvBuffer);
      wrap_bufs[1].pvBuffer = NULL;

      /* read unencrypted server response */
      n = readline(sockfd, recvline, MAXLINE);
      if (n < 0) err_dump("str_cli: readline error");
      recvline[n] = 0;
      fputs(recvline, stdout);
   }
   if (ferror(stdin)) err_sys("str_cli: error reading file");

}

int send_token(int s, PSecBuffer tok) {
   ULONG          len;
   ULONG          ret;
   unsigned char  length_low;
   unsigned char  length_high;
```

```
   len = htonl(tok->cbBuffer);
   length_low =  tok->cbBuffer  & 0xff;
   length_high = tok->cbBuffer  >> 8;

   ret = writen(s, (char *) &length_low, 1);
   if (ret < 0) {
      perror("sending token length");
      return -1;
   }
   ret = writen(s, (char *) &length_high, 1);
   if (ret < 0) {
      perror("sending token length");
      return -1;
   }

   ret = writen(s, tok->pvBuffer, tok->cbBuffer);
   if (ret < 0) {
      perror("sending token data");
      return -1;
   }
   else if (ret != tok->cbBuffer) {
      fprintf(stderr,
        "sending token data: %d of %d bytes written\n",
        ret, tok->cbBuffer);
      return -1;
   }

   return 0;
}


int recv_token(int s, PSecBuffer tok) {
   ULONG          ret;
   unsigned char  length_low;
   unsigned char  length_high;

   ret = readn(s, (char *) &length_low, 1);
   if (ret < 0){
      perror("reading token length");
      return -1;
   }
   ret = readn(s, (char *) &length_high, 1);
   if (ret < 0) {
      perror("reading token length");
      return -1;
   }

   tok->cbBuffer  = length_low + length_high * 256;
```

```
   tok->pvBuffer = (char *) malloc(tok->cbBuffer);
   if (tok->pvBuffer == NULL) {

      fprintf(stderr,
                   "Out of memory allocating token data\n");
      return -1;
   }

   ret = readn(s, (char *) tok->pvBuffer, tok->cbBuffer);
   if (ret < 0) {
      perror("reading token data");
      free(tok->pvBuffer);
      return -1;
   }
   else if (ret != tok->cbBuffer) {
      fprintf(stderr,
              "sending token data: %d of %d bytes written\n",
              ret,
              tok->cbBuffer);
      free(tok->pvBuffer);
      return -1;
   }

   return 0;
}

#endif
```

Header file for utility source

Description: Utility module without DCE dependencies
File name: utils_s1.h

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>

#ifdef WIN32
#include <windows.h>
#include <io.h>
#include <fcntl.h>
#include <winsock2.h>
#define SECURITY_WIN32
#include <security.h>
#include <ntsecapi.h>
#include "gssapi/gssapi.h"

/* For some reason this is not visible through include, so we define it here */
```

```
# define SECBUFFER_PADDING 9

# undef write
# define write(fd, bufptr, buflen) send(fd, bufptr, buflen, 0)
# undef read
# define read(fd, bufptr, buflen) recv(fd, bufptr, buflen, 0)
# undef close
# define close(fd) closesocket(fd)

#else
# include <gssapi/gssapi.h>
# include <sys/socket.h>
# include <netinet/in.h>
# include <arpa/inet.h>
# include <netdb.h>
#define min(a,b) (((a)<(b))?(a):(b))
#endif

#define err_sys(x) {perror(x); exit (1);}
#define err_dump(x) {fprintf(stderr, x); exit (1);}

int readn(register int fd,
      register char * ptr,
      register int nbytes);

int writen(register int fd,
       register char * ptr,
       register int nbytes);

int readline(register int fd,
         register char * ptr,
         register int maxlen);
#ifdef WIN32
void str_cli(register FILE *fp,
             register int sockfd,
             CtxtHandle  *security_context);
#else
void str_s1(int sockfd, gss_ctx_id_t security_context);
#endif
int read_token(int socket, gss_buffer_t tok);
int write_token(int socket, gss_buffer_t tok);
void display_error(char * where, OM_uint32 maj, OM_uint32 min);
```

Application configuration file

Description: Application configuration file for IBM Tivoli Access Manager access
File name: remote.conf

```
#
# Licensed Materials - Property of IBM
# 5748-XX8
# (c) Copyright International Business Machines Corp. 2001
# All Rights Reserved
# US Government Users Restricted Rights - Use, duplicaion or disclosure
# restricted by GSA ADP Schedule Contract with IBM Corp.
#
# File Name: remote.conf
#*********************************************************************
#
#               Example configuration file for the aznAPI application.
#
#*********************************************************************
#
# Each stanza in this file has a separate section for parameters
# that apply for local, remote, or both modes of operation. The
# section is omitted if the stanza in question does not have
# parameters that apply only to it.
#
# Parameters not specified as being set by svrsslcfg must be set by
# editing this file and providing the required values.
#
[aznapi-configuration]

#*********************************************************************
#
#                  aznAPI Configuration Settings -- Local Mode
#
#*********************************************************************
#
# NOTE: The following parameters only apply for local mode.
# For remote mode, these parameters will be ignored.
# To enable the feature in question for remote mode, the corresponding
# parameter needs to be set in the pdacld configuration file  --
# ivacld.conf.
#
#*********************************************************************
#
# Audit and Logging parameters.
#
#       logsize         (in bytes) The size at which the log files will
#                       rollover to a new file. If set to 0 the log files
#                       will not rollover. A negative number will roll
```

```
#                       the logs over daily regardless of size.
#       logflush        The interval in seconds at which the logs are
#                       flushed. Maximum of 6 hours and a default of 20
#                       seconds.
#
logsize = 2000000
logflush = 20

# Audit logging configuration.
#
#       logaudit        Boolean. To turn on/off auditing completely.
#       auditlog        the name of the audit file.
#       auditcfg        To disable or enable component specific audit
#                       records you may add or remove the appropriate
#                       'auditcfg' definition.
#
logaudit = no
auditlog = audit.log
auditcfg = azn
#auditcfg = authn

# The set of attributes that the caller is interested in receiving
# from the azn_decision_access_allowed_ext() call in the permission
# info attribute list. By default there is no information returned
# by this call. Adding an attribute name to this list will enable
# the attribute to be returned as permission info if it is applicable
# to the current decision call. This list may also include attributes
# that are user defined. e.g. by setting an attribute on an ACL using
# the "acl modify set attribute" command from "pdadmin".
#
# The following string constants are recognised by the authzn engine
# and equate to their corresponding permission info attribute
# constants in ogauthzn.h. Refer to this file for more information on
# these permission info attributes:
#
#       azn_perminfo_all_attrs          Return all attributes.
#       azn_perminfo_al_new             Audit level (uint).
#       azn_perminfo_qop                Quality of Protection (string).
#       azn_perminfo_qop_uint_new       Quality of Protection (uint).
#       azn_perminfo_wm_new             Warning mode (bool).
#       azn_perminfo_wm_permitted_new   Access permitted by warning
#                                       mode (bool).
#
# e.g.:
#    permission-info-returned = azn_perminfo_qop_uint_new my_attribute
#
#permission-info-returned = azn_perminfo_qop azn_perminfo_qop_uint_new

#
```

```
# The path for the ACL database cache file.
#
db-file = ./authzn_demo.db

# Update poll interval. This is the interval, in seconds, between
# checks for updates to the master authzn server. The local cache is
# only rebuilt if an update is detected. Values can be "disable",
# "default" or a time in seconds.
#
cache-refresh-interval = disable

#
# Flags to enable the reception of policy cache update notifications.
# Values can be one of: "disable", "enable"
# A "disable" value the notification listener.
#
# This parameter is set by the svrsslcfg utility.
#
listen-flags = disable

# azn-app-host:  This attribute is used to customize the host on which the
#                AZN application is listening.  If this attribute is not
#                specified, the default hostname is used.  To enable this
#                attribute, create a new line similar to the following:
# azn-app-host = <other hostname>

#**********************************************************************

#**********************************************************************
#
#                 aznAPI Configuration Settings -- Common
#
#**********************************************************************
#
# NOTE: The following parameters apply for both local and remote mode.
#
#**********************************************************************
#
# Start-up parameters.  These are used for initialising the API.
# Typically there is no need to change these from the defaults.
#

# Operating mode for the AuthAPI.  Values are
#
#     remote           Uses ivacld.
#     local            Starts an API client with a local policy cache.
#
#  This parameter is set by svrsslcfg utility
```

```
mode = remote




azn-server-name = authzn_remote-tivdce1.itsc.austin.ibm.com
pd-user-name = authzn_remote/tivdce1.itsc.austin.ibm.com
pd-user-pwd = chuy5
[ssl]

#*******************************************************************
#
#                 ssl Configuration Settings -- Common
#
#*******************************************************************
#
# NOTE: The following parameters apply for both local and remote mode.
#
#*******************************************************************
#
#
# SSL configuration
#     The svrsslcfg utility will set most of these parameters.
#

#
# Pathname on the local system for the keyfile used by SSL.
#
ssl-keyfile = /home/root/devel/keytab/authzn_remote.kdb

#
# File containing the password used to protect private keys in the
# keyfile.  The password is obfuscated and stored in the stash file.
#
ssl-keyfile-stash = /home/root/devel/keytab/authzn_remote.sth

#
# NOTE: The following parameter only applies for these scenarios:
#       * local mode when listen-flags are set to enable.
#       * local mode when aznapi-admin-services are registered.
#       * remote mode when aznapi-admin-services are registered.
# For other cases, this parameter will be ignored.
#
# Port on which the application will listen for requests and policy
# cache update notifications.
#
ssl-listening-port = 7777

#
```

```
# Maximum number of threads that will be created by the aznAPI's
# internal server to handle incoming requests (range: >= 1, optimal
# limits are system resource dependent).
#
ssl-maximum-worker-threads = 10

#
# Session time-out in seconds for SSL v3 connections (range: 1-86400).
#
ssl-v3-timeout = 7200

#
# Connection time-out for i/o in seconds
# (range: >= 0, 0 = no time-out).
#
ssl-io-inactivity-timeout = 90

#
# Key database file password lifetime in days.
#
ssl-pwd-life = 183

#
# Enable (value of "yes") or disable (value of "no") automatic refresh
# of the SSL certificate and key database file password.  When
# enabled, the certificate and password will be regenerated if either
# is in danger of expiration.
#
ssl-auto-refresh = yes

#
# Authentication type.
#
ssl-authn-type = certificate

[ldap]

#*******************************************************************
#
#                 LDAP Configuration Settings -- Common
#
#*******************************************************************
#
# NOTE: The following parameters apply for both local and remote mode.
#
#*******************************************************************
#
# Configure the LDAP user registry interface if required.
#
```

```
# If enabled then you will require an LDAP server host name, a port
# to bind to the server on, a bind user DN and bind user password.
# Optionally, you can specify that the aznAPI communicate with the
# server using SSL. In this case you must set the enable parameter
# and you must also specify an SSL keyfile name to use along with the
# keyfile DN (if there are multiple keys in the file) and keyfile DN's
# password.
#
#
# LDAP config parameters.
#
#    enabled                    Enable LDAP user registry support? Yes/no.
#                               This parameter will be set by svrsslcfg utility
#                               depending on the configured Policy Director User
#                               Registry type.
#    host                       The LDAP server host name.
#                               This parameter will be set by svrsslcfg utility
if
#                               the configured Policy Director User Registry is
LDAP.
#    port                       The IP port on which the LDAP server listens for
non-SSL
#                               requests.
#    bind-dn                    An LDAP user DN to bind to the server with.
#                               This parameter will be set by svrsslcfg utility
if
#                               the configured Policy Director User Registry is
LDAP.
#    bind-pwd                   The pwd for the above LDAP user DN.
#                               This parameter will be set by svrsslcfg utility
if
#                               the configured Policy Director User Registry is
LDAP.
#
#    cache-enabled              Enable LDAP client-side caching to improve
#                               performance for similar LDAP queries. True/false.
#
#    ssl-enabled                (optional) Enable SSL comms with the LDAP server.
#                               Yes/no.
#    ssl-port                   The SSL IP port on which the LDAP server listens
#                               for SSL requests.  Required if ssl-enabled = yes
#    ssl-keyfile               Path to an SSL key file containing the LDAP server
#                               certificate.  Required if ssl-enable = yes
#    ssl-keyfile-pwd            Password for the key file.
#                               Required if ssl-enable = yes
#    ssl-keyfile-dn             Key file DN (for files with multiple keys).
#
#    max-search-size            (optional) Limit for the maximum search buffer
#                               size returned from the LDAP server in entries.
```

```
#
#    prefer-readwrite-server  (optional) The client will attempt to query
#                             the read/write LDAP server (see ldap-replica
#                             config option below) before any read-only
#                             servers configured in the domain.
#
#    auth-using-compare       (optional) Choose whether ldap_compare() will
#                             be used instead of the ldap_bind() call to
#                             authenticate LDAP users. This option will
#                             change the method used by these aznAPI calls:
#
#                                 - azn_util_client_authenticate()
#                                 - azn_util_password_authenticate()
#

enabled = yes
host = tivdce1.itsc.austin.ibm.com
port = 389
bind-dn =
cn=authzn_remote/tivdce1.itsc.austin.ibm.com,cn=SecurityDaemons,secAuthority=De
fault
bind-pwd = chuy5
cache-enabled = true

#ssl-enabled = yes
#ssl-port = 636
#ssl-keyfile = <key file>
#ssl-keyfile-pwd = <key file password>
#max-search-size = 2048
#prefer-readwrite-server = true
#auth-using-compare = true


# Define the LDAP user registry replicas in the domain.
#
# Each string is of the format:
#   ldap-replica = <ldap-server>,<port>,<type>,<pref>
#
# Where:
#   <ldap-server> is the network name of the server.
#   <port> is the port on the ldap server.
#   <type> is one of "readonly" or "readwrite"
#   <pref> is a level from 1 to 10.  10 is the highest preference.
#
#ldap-replica = freddy,390,readonly,1
#ldap-replica = barney,391,readwrite,2
#ldap-replica = benny,392,readwrite,3
```

```
[uraf-ad]

#*********************************************************************
#
#                  Active Directory Configuration Settings
#
#*********************************************************************

ad-server-config = <Active Directory registry configuration file for PD>
bind-id = <server identity for bind>
bind-pwd = <server password for bind>

[uraf-domino]

#*********************************************************************
#
#                  Domino Configuration Settings
#
#*********************************************************************

domino-server-config = <Domino registry configuration file for PD>
bind-id = <server identity for bind>
bind-pwd = <server password for bind>

#*********************************************************************
#
#                     aznAPI Service Definitions - Common
#
#*********************************************************************
#
# NOTE:  The following aznAPI service definitions apply for both local
# and remote mode.
#
#*********************************************************************
#
# aznAPI service definitions.
#
# Each stanza entry defines different types of aznAPI service.
# For more information refer to the Authorization API programmers
# guide.
#
# Each entry is of the format:
#
#       <service-id> = <path-to-dll> [ & <params> ... ]
#
# e.g.
#       AZN_ENT_SVC_USER = /usr/lib/libazn_ent_user.so & -param 1
#
# The <service-id> is the string by which the service can be
```

```
# identified by the aznAPI client. The <path-to-dll> is the
# path to the DLL which contains the service executable code.
# If the DLL resides in a directory that is normally searched
# by the system for DLLs e.g. /usr/lib on Unix platforms and
# %PATH% on Windows-NT then you need not specify the full path
# to the DLL, simply the DLL name. If you want the DLL name
# to be platform independent so that it may be loaded on any
# supported Policy Director platform then you may provide a
# short form library name. The short name will be prepended
# and appended with known library prefixes and suffixes for
# each platform and each possibility searched for in turn.
# e.g. with an example short form library name of "azn_ent_user"
# then the following names will be automatically searched for
# on each platform:
#
#    NT:      azn_ent_user.dll
#    AIX:     libazn_ent_user.so, libazn_ent_user.a
#    Solaris: libazn_ent_user.so
#    HP/UX:   libazn_ent_user.sl
#
# Optionally you can specify parameters to pass to the service
# when it is initialised by the aznAPI. The parameters are
# considered to be all data following the '&' symbol in the
# service definition.
#
# Refer to the Authorization Programmer's Guide for any
# differences in the way that the service definition for each
# particular service type may vary from that above.
#



[aznapi-entitlement-services]



[aznapi-pac-services]



[aznapi-cred-modification-services]



[aznapi-external-authzn-services]



[aznapi-admin-services]
```

```
#
# Sample AZN Admin. Service Definitions
#
# plugin w/ pobj parameter, and with plugin parameters
#AZN_ADMIN_SVC_DEMO1 = azn_admin_svc_demo -pobj
/aznadminsvc/authzn_demo_server1 & -server authzn_demo_server1
#
# plugin w/ no pobj parameter, and with plugin parameters
#AZN_ADMIN_SVC_DEMO2 = azn_admin_svc_demo & -server authzn_demo_server2
#

AZN_ADMIN_SVC_TRACE = pdtraceadmin



[manager]

#********************************************************************
#
#                  manager Configuration Settings -- Remote
#
#********************************************************************
#
# NOTE:  The following parameters apply only for remote mode.
# For local mode, this parameter will be ignored.
#
#********************************************************************
#
#
# Authorization server replicas
#
#  Use svrsslcfg utility -add_replica to add replica entries to this
#  file.
#
#  replica = \
#      <ivacld replica hostname>:<port>:<preference>:<replica cert dn>
#
# For example,
#  replica = \
#  rep1.acme.com:7137:5:cn=ivacld/rep1.acme.com,o=Policy Director,C=US

#********************************************************************
#
#                  manager Configuration Settings -- Common
#
#********************************************************************
#
# NOTE: The following parameters apply for both local and remote mode.
```

```
#
#*********************************************************************
#
#
# PD Management server configuration
#  svrsslcfg utility  will set parameters in this section
#

# Hostname of the Management server
master-host = tivdce2.itsc.austin.ibm.com

# TCP Port number on which the server is listening for requests.
master-port = 7135



master-dn = CN=ivmgrd/master,O=Policy Director,C=US
replica =
tivdce1.itsc.austin.ibm.com:7136:10:CN=ivacld/tivdce1.itsc.austin.ibm.com,O=Pol
icy Director,C=US
[authentication-mechanisms]

#*********************************************************************
#
#         authentication-mechanisms Configuration Settings -- Common
#
#*********************************************************************
#
# NOTE: The following parameters apply for both local and remote mode.
#
#*********************************************************************
#
#
# Authentication mechanisms
# Entries in this stanza are managed by svrsslcfg utility
passwd-ldap = /opt/PolicyDirector/lib/libldapauthn.a
cert-ldap = /opt/PolicyDirector/lib/libcertauthn.a
```

B

Scenario 2: Source code listings

This appendix lists the complete source code that was explained in pieces in Chapter 9, “Scenario 2: Non-secure RPC application” on page 171. Specifically, this appendix contains:

- The source code of the client and server programs with DCE dependencies
- The source code of the revised client and server program without DCE dependencies
- The makefiles used for these programs
- Header files
- Application properties files

Please refer to Appendix E, “Additional material” on page 417 for instructions about downloading the source code samples included in this appendix.

© Copyright IBM Corp. 2003. All rights reserved.

Application with DCE dependencies

The files are:

- Makefile contains information about how the application is compiled and linked on the AIX operating system.
- client_s2.mak contains information about how the application is compiled and linked on the Windows operating system.
- greet.idl contains the interface definition shared by the client and server programs. It contains a single service: greet_rpc()
- client_s2.c contains the client code. After acquiring a partial binding to the server from the DCE CDS naming service, the client program invokes an RPC to the server and prints the server's response to the screen.
- server_s2.c contains the server code. The server creates and registers its binding information with DCE CDS Naming Service and DCE RPC endpoint mapper. Then it starts listening for client calls.
- server_manager.c contains the actual application logic of the server program. Upon client requests, it prints the client's string message to the screen and returns another string to the client.

The source files greet_cstub.c, greet_sstub.c, and greet.h are generated by the DCE IDL compiler so are not listed here.

Makefile for the AIX platform

Description: Makefile for application client and server
File name: Makefile.aix

```
SOURCE = server_s2.c client_s2.c server_manager.c

SOBJS = server_s2.o server_manager.o greet_sstub.o
COBJS = client_s2.o greet_cstub.o

IDL = /usr/bin/idl

IDL_INCDIRS = -I..

IDL_FLAGS = ${IDL_INCDIRS} -keep c_source

INCDIRS = -I.

LIBDIRS = -L/usr/lib -L.

LIBS = ${LIBDIRS} -ldce
```

```
CDEFS = -Dunix -qlanglvl=extended -D_ALL_SOURCE
CFLAGS  = -g -w ${CDEFS} ${INCDIRS}

CC = /bin/xlc_r4

.c.o:
   ${CC} ${CFLAGS} -c $<

all:server_s2 client_s2

server_s2.o:  server_s2.c greet.h
   ${CC} ${CFLAGS} -c server_s2.c

client_s2.o:  client_s2.c greet.h
   ${CC} ${CFLAGS} -c client_s2.c

server_manager.o: server_manager.c greet.h
   ${CC} ${CFLAGS} -c server_manager.c

greet_sstub.o: greet_sstub.c greet.h
   ${CC} ${CFLAGS} -c greet_sstub.c

greet_cstub.o: greet_cstub.c greet.h
   ${CC} ${CFLAGS} -c greet_cstub.c

greet_cstub.c greet_sstub.c greet.h: greet.idl
   ${IDL} greet.idl ${IDL_FLAGS}

server_s2: ${SOBJS}
   ${CC} -o server_s2 ${SOBJS} ${LIBS}

client_s2: ${COBJS}
   ${CC} -o client_s2 ${COBJS} ${LIBS}

clean:
   rm -f greet.h greet_sstub.c greet_cstub.c ${SOBJS} ${COBJS}

rmtarget:
   rm -f server_s2 client_s2

clobber: clean rmtarget
```

Makefile for the Windows platform

Description: Makefile for application client
File name: greet.mak

```
!include <NTWIN32.MAK>

!if "$(CPU)" == "ALPHA"
TARGET=alpha
SWITCHES=-DALPHA
!else
TARGET=intel
SWITCHES=-D_X86=1 -DM_I86
!endif

_LIBS_ = \
   libdce.lib \
   pthreads.lib \
   $(conlibsdll)

IDL = idl -keep c_source -v

CFLAGS = -I. $(cflags:-W3=) $(cvarsdll) $(SWITCHES) -Zi -Od -Gz -nologo

LINK = link $(ldebug) $(conlflags)

all: client_s2.exe server_s2.exe

clean :
   -del *.obj
   -del *.exe
   -del greet_cstub.c
   -del greet_sstub.c
   -del greet.h

client_s2.exe : client_s2.obj greet_cstub.obj
   $(LINK) /out:$@ $** $(_LIBS_)

server_s2.exe : server_s2.obj server_manager.obj greet_sstub.obj
   $(LINK) /out:$@ $** $(_LIBS_)

greet_cstub.obj : greet_cstub.c greet.h
   $(CC) $(CFLAGS) greet_cstub.c

server_s2.obj : server_s2.c greet.h
   $(CC) $(CFLAGS) server_s2.c

server_manager.obj : server_manager.c greet.h
   $(CC) $(CFLAGS) server_manager.c
```

```
client_s2.obj : client_s2.c greet.h
   $(CC) $(CFLAGS) client_s2.c

greet_sstub.obj : greet_sstub.c greet.h
   $(CC) $(CFLAGS) greet_sstub.c

greet_cstub.c greet_sstub.c greet.h : greet.idl
   $(IDL) -client stub -server stub  -cstub greet_cstub.c \
      -sstub greet_sstub.c greet.idl
```

IDL source

Description: Interface definition
File name: greet.idl

```
[
uuid(d58ab008-b3c6-11ca-891c-c9c2d4ff3b52),
version(1.0)
]

interface greet
{
      const long int STR_SZ = 128;

      void greet_rpc (
           [in]  handle_t h,
           [in,string]  char client_greeting[STR_SZ],
           [out,string] char server_reply[STR_SZ]
       );
}
```

DCE dependent application client

Description: Application client with DCE dependencies
File name: client_s2.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <dce/dce_error.h>
#include "greet.h"


int main(int ac, char*av[]) {
   handle_t               h;
   error_status_t         st;
```

```
idl_char                 *string_binding;
int                      i, MAX_PASS, error_inq_st;
char                     reply[STR_SZ], error_string[1024],
                         *name_to_greet;
rpc_ns_import_handle_t   import_context;

if (ac != 3)  {
    fprintf(stderr, "Usage: %s message passes\n", av[0]);
    exit (1);
}

name_to_greet = av[1];
printf("Name to greet: %s\n",name_to_greet);

/* import compatible server bindings from the namespace */

rpc_ns_binding_import_begin(rpc_c_ns_syntax_dce,
                            "/.:/servers/greet", greet_v1_0_c_ifspec,
                           (uuid_t *)NULL, &import_context, &st);

if (st != error_status_ok) {
    dce_error_inq_text(st, error_string, &error_inq_st);
    fprintf(stderr, "Cannot begin importing binding - %s\n", error_string);
    exit(1);
   }

/* sift through bindings and choose the first one over udp */

while (1) {
    rpc_ns_binding_import_next(import_context, &h, &st);
    if (st == rpc_s_no_more_bindings) {
       dce_error_inq_text(st, error_string, &error_inq_st);
       fprintf(stderr, "Cannot find binding over udp: %s\n", error_string);
       exit(1);
    }

     rpc_binding_to_string_binding(h, &string_binding, &st);
     if (st != error_status_ok) {
        dce_error_inq_text(st, error_string, &error_inq_st);
        fprintf(stderr, "Cannot convert binding to string binding: %s\n",
        error_string);
        exit(1);
     }

     /* out of curiosity, print the binding */

     if (strstr(string_binding, "ncadg_ip_udp") != 0) {
         fprintf(stdout, "Client bound to server at %s\n", string_binding);
         rpc_string_free(&string_binding, &st);
```

```
            break;
        }

    }
    /* end the binding import lookup loop */

    rpc_ns_binding_import_done(&import_context, &st);
    if (st != error_status_ok) {
        dce_error_inq_text(st, error_string, &error_inq_st);
        fprintf(stderr, "Cannot end binding import: %s\n", error_string);
        exit(1);
    }

    fprintf(stdout, "\n");

    MAX_PASS= atoi(av [2]);

    for (i=1; i <= MAX_PASS; i++) {
        greet_rpc(h, name_to_greet, reply);
        printf("The Greet Server said: %s\n", reply);
        fflush(stdout);
    }
    return(0);
}
```

DCE dependent application server

Description: Application server with DCE dependencies
File name: server_s2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <dce/pthread_exc.h>
#include <dce/dce_error.h>
#include <dce/rpc.h>
#include "greet.h"

#define MAX_CONCURRENT_CALLS 5

/* The server declares nil UUID which it supplies
   in registrations as an object UUID and object type UUID. */

#ifdef __BORLANDC__
extern uuid_t _pascal uuid_nil;
#else
extern uuid_t uuid_nil;
#endif
```

```
/* In the first part of the main function, the server calls
   the rpc_network_is_protseq_valid to check that its argument
   specifies a protocol sequence that is supported on its host
   both by the runtime library and the operating system.  */

int main () {
    rpc_binding_vector_p_t  bvec;
    error_status_t          st;
    boolean32               validfamily;
    idl_char                *string_binding;
    char                    *entry_name, error_string[1024];
    int                     i, error_inq_st;

    entry_name = "/.:/servers/greet";

    validfamily = rpc_network_is_protseq_valid("ncadg_ip_udp", &st);
    if (st != error_status_ok) {
        dce_error_inq_text (st, error_string, &error_inq_st);
        fprintf (stderr,
         "Cannot check protocol sequence - %s\n",
         error_string);
        exit(1);
    }

    if (!validfamily)  {
        fprintf(stderr, "Protocol sequence is not valid\n");
        exit (1);
    }

    /* Calling rpc_server_use_protseq to obtain an endpoint
       on which to listen */

    rpc_server_use_protseq("ncadg_ip_udp", MAX_CONCURRENT_CALLS, &st);
    if (st != error_status_ok) {
        dce_error_inq_text (st, error_string, &error_inq_st);
        fprintf (stderr, "Cannot use protocol sequence - %s\n", error_string);
        fflush(stdout);
        exit(1);
    }


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

    rpc_server_register_if(greet_v1_0_s_ifspec, (uuid_p_t)NULL, NULL, &st);
    if (st != error_status_ok) {
        dce_error_inq_text (st, error_string, &error_inq_st);
        fprintf (stderr, "Cannot register interface - %s\n", error_string);
```

```
    fflush(stdout);
    exit(1);
}

/* Calling rpc_server_inq_bindings to obtain a vector of
   binding handles that can be used to register the server's
   endpoint. The server then obtains, prints, and frees a
   string binding */

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

printf("Server %s bindings:\n", entry_name);
fflush(stdout);

for (i = 0; i < bvec->count; i++)  {
     rpc_binding_to_string_binding(bvec->binding_h[i],
          &string_binding, &st);

     printf("%s\n", (char *)string_binding);
     fflush(stdout);
     rpc_string_free(&string_binding, &st);

}

/* The server endpoint is registered in the local Endpoint Map */

rpc_ep_register(greet_v1_0_s_ifspec, bvec,
                (uuid_vector_p_t) NULL,
                (unsigned_char_p_t) "greet version 1.0 server",
                &st);
if (st != error_status_ok) {
      dce_error_inq_text (st, error_string, &error_inq_st);
      fprintf (stderr, "Cannot register end point: %s\n", error_string);
      fflush(stdout);
      exit(1);
}


/* export the binding vector the runtime gave us to the namespace */

rpc_ns_binding_export(rpc_c_ns_syntax_dce,
                      (unsigned_char_p_t) "/.:/servers/greet",
                      greet_v1_0_s_ifspec, bvec,
```

```
                        (uuid_vector_t *)NULL, &st);

if (st != error_status_ok) {
   dce_error_inq_text(st, error_string, &error_inq_st);
   fprintf(stderr, "Cannot export binding vector: %s\n",error_string);
   fflush(stdout);
   exit(1);
}

/*  To begin listening for RPC requests, the server calls
  rpc_server_listen.  This call is placed within the TRY of a
  TRY, CATCH_ALL, ENDTRY sequence, so that if the server receives
  a signal while it is listening, it can unregister its interface
  and its endpoint before it exits.   */

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

CATCH_ALL {

   /* unexport binding vector from namespace --
      not usually done for a persistent server */

   fprintf(stdout, "Server %s unexporting\n", entry_name);
   fflush(stdout);

   rpc_ns_binding_unexport(rpc_c_ns_syntax_dce,
                        "/.:/servers/greet", greet_v1_0_s_ifspec,
                        (uuid_vector_t *)NULL, &st);

   if (st != error_status_ok) {
      dce_error_inq_text(st, error_string, &error_inq_st);
      fprintf(stderr, "Cannot unexport binding vector - %s\n",
              error_string);
      fflush(stdout);
      /* don't exit here */
    }

     printf("Unregistering endpoint \n");
     fflush(stdout);

     rpc_ep_unregister(greet_v1_0_s_ifspec, bvec,
```

```
        (uuid_vector_p_t) NULL, &st);
    }
    ENDTRY;
    return (0);
}
```

Application server logic

Description: Application server logic
File name: server_manager.c

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

#ifndef IBMOS2
#define DCEAPI
#endif

/* server application logic :-) */
void greet_mirror(char * client_greeting, char * server_reply) {
   char tmp[STR_SZ];
   int i,msglen;

   msglen = strlen(client_greeting);
   printf("The client says: %s\n", client_greeting);
   for (i=0;i<msglen;i++) {
     tmp[i] = client_greeting[msglen-i-1];
   }
   tmp[i] = '\0';
   printf("This I turn around into %s\n",tmp);
   fflush(stdout);
   strncpy(server_reply,tmp,msglen+1);

}


void  DCEAPI greet_rpc(
/* [in] */  handle_t h,
/* [in] */  idl_char client_greeting[128],
/* [out] */ idl_char server_reply[128] ) {

   greet_mirror(client_greeting, server_reply);

}
```

Revised application without DCE dependencies

The files are:

- *GreetAi*x.mak contains information about how the application is compiled and linked on the AIX operating system.
- GreetNt.mak contains information about how the application is compiled and linked on the Windows operating system.
- greet.idl contains the interface defintion shared by the client and server programs. It contains a single service: greet_CORBA()
- CWrapper.h contains the function prototypes for all of the C wrapper functions.
- client_CORBA.cpp contains the CORBA C++ client code. It initializes the client environment, gets a pointer to the root naming context, accesses the servant object, calls methods on the servant object, and stops the client and releases resources used.
- client_s2.c contains the C client code. After intializing the CORBA environment by invoking a C wrapper function defined in the CORBA C++ client, it invokes another C wrapper function to access the servant method and prints the server's response to the screen.
- greet.ih is the header file for the servant implementation class. It has the class variables and methods declaration. This contains the modifications made to the header file after it is generated by the CORBA IDL compiler.
- greet_I.cpp is the servant implementation class. It has method definitions for the declarations in the greet.ih file. This contains the modifications made to the implementation file after it is generated by the CORBA IDL compiler.
- server_CORBA.cpp contains the CORBA C++ server code. It initializes the server environment, accesses naming contexts, binds a servant object, creates a server shutdown object, and then starts listening for client calls to service client requests.
- server_manager.c contains the actual application logic of the server program. Upon client requests, it prints the client's string message to the screen and places the reserved message in the reply.
- WSServer.props contains the run-time properties for the CORBA C++ server.
- WSClient.props contains the run-time properties for the CORBA C++ client.

The source files greet_C.cpp, greet_S.cpp, and greet.hh are generated by the CORBA IDL compiler and therefore not listed here.

Makefile for the AIX platform

Description: Makefile for the revised application client and server
File name: GreetAix.mak

```
#AIX-Specific Variables
IDLC   = $(WAS_HOME)/bin/idlc
REMOVE= /bin/rm
REMOVEDIR= /bin/rm -rf
COPY   = /bin/cp
CCC    = /usr/vacpp/bin/xlC_r

#WAS-Specific Variables
LIBPATH= $(WAS_HOME)/lib
INCDIRS= $(WAS_HOME)/include

#OLT Debug Specific variables
LOCAL_CCFLAGS_DEBUG = -g
LOCAL_LDFLAGS_DEBUG = -g

#Other Variables
WSLOGDIR        = .
LOCAL_INCDIRS= -I$(WSLOGDIR)

LIBS   = $(LIBPATH) -L$(WSLOGDIR) -L/usr/vacpp/lib
INCS   = $(INCDIRS) $(LOCAL_INCDIRS) -I/usr/vacpp/include

LOCAL_CCFLAGS   = -c $(LOCAL_CCFLAGS_DEBUG) -M -DEXCL_IRTC -D_USE_NAMESPACE
-D_UNIX
LOCAL_LDFLAGS   = $(LOCAL_LDFLAGS_DEBUG) -T512 -H512 -lwasoror -lwasosa1
-lwassrvs -lC

.SUFFIXES: .o .c .cpp .java .class

all: GreetServer GreetClient

GreetServer: Server_CORBA.o server_manager.o greet_I.o greet_S.o
   $(CCC) -o $@ $? -L$(LIBS) $(LOCAL_LDFLAGS)

GreetClient: client_s2.o client_CORBA.o
   $(CCC) -o $@ $? -L$(LIBS) $(LOCAL_LDFLAGS)

Server_CORBA.o: server_manager.c Server_CORBA.cpp greet_S.cpp greet_I.cpp
   $(CCC) $(LOCAL_CCFLAGS) -I$(INCS) $?

client_s2.o: client_s2.c
   $(CCC) $(LOCAL_CCFLAGS) -I$(INCS) $?

client_CORBA.o: client_CORBA.cpp greet_C.cpp
```

```
   $(CCC) $(LOCAL_CCFLAGS) -I$(INCS) $?

greet_S.o: greet_S.cpp
   $(CCC) $(LOCAL_CCFLAGS) -I$(INCS) $?

greet_I.o: server_manager.c greet_I.cpp
   $(CCC) $(LOCAL_CCFLAGS) -I$(INCS) $?

greet_C.o: greet_C.cpp
   $(CCC) $(LOCAL_CCFLAGS) -I$(INCS) $?

greet_S.cpp: greet.idl
   $(IDLC) -J"-Djava.ext.dirs=$(EXTDIRS)" -mcpponly \
   -shh:uc:sc greet.idl

clean:
   -$(REMOVE) GreetServer GreetClient \
   *.o *.u \
   greet_C.cpp greet_S.cpp \
   greet.hh
```

Makefile for the Windows platform

Description: Makefile for the revised application client and server
File name: GreetNt.mak

```
#Windows-Specific Variables
IDLC   = idlc
ILIB   = lib
REMOVE= del
COPY   = copy
CCC    = cl
LINK   = link

#WAS-Specific Variables
LIBPATH= $(WAS_HOME)\lib
INCDIRS= $(WAS_HOME)\include

#OLT Debug Specific Variables
!ifdef DEBUG_ON
LOCAL_CCFLAGS_DEBUG = /Z7
LOCAL_LDFLAGS_DEBUG = /DEBUG /PDB:NONE
!else
LOCAL_CCFLAGS_DEBUG =
LOCAL_LDFLAGS_DEBUG =
!endif
```

```
#Other Variables
WSLOGDIR= .
EXTDIRS= $(WSLOGDIR);$(WAS_HOME)\lib
LOCAL_INCDIRS= $(WSLOGDIR)
LOCAL_CCFLAGS= -DEXCL_IRTC -D_USE_NAMESPACE \
               -GX $(LOCAL_CCFLAGS_DEBUG) /c /nologo /MD /Od /Fm
LOCAL_LDFLAGS= /NOLOGO $(LOCAL_LDFLAGS_DEBUG) /LIBPATH:$(LIBS) \
               wasororm.lib wasosa1m.lib wassrvsm.lib

LIBS  = $(LIBPATH)
INCS  = $(INCDIRS) /I$(LOCAL_INCDIRS)
SOBJS = Server_CORBA.obj server_manager.obj greet_I.obj greet_S.obj
COBJS = client_CORBA.obj client_s2.obj greet_C.obj

.SUFFIXES: .obj .c .cpp .java .class

all: GreetServer1.exe GreetClient.exe

GreetServer1.exe: Server_CORBA.obj server_manager.obj greet_I.obj greet_S.obj
   $(LINK) $(LOCAL_LDFLAGS) /OUT:GreetServer1.exe $(SOBJS)

GreetClient.exe: client_s2.obj client_CORBA.obj greet_C.obj
   $(LINK) $(LOCAL_LDFLAGS) /OUT:GreetClient.exe $(COBJS)

Server_CORBA.obj: server_manager.c Server_CORBA.cpp greet_S.cpp greet_I.cpp
   $(CCC) /I$(INCS) $(LOCAL_CCFLAGS) $?

client_s2.obj: client_s2.c
   $(CCC) /I$(INCS) $(LOCAL_CCFLAGS) $?

client_CORBA.obj: client_CORBA.cpp greet_C.cpp
   $(CCC) /I$(INCS) $(LOCAL_CCFLAGS) $?

greet_S.obj: greet_S.cpp
   $(CCC) /I$(INCS) $(LOCAL_CCFLAGS) $?

greet_I.obj: server_manager.c greet_I.cpp
   $(CCC) /I$(INCS) $(LOCAL_CCFLAGS) $?

greet_C.obj: greet_C.cpp
   $(CCC) /I$(INCS) $(LOCAL_CCFLAGS) $?

greet_S.cpp: greet.idl
   $(IDLC) -J"-Djava.ext.dirs=$(EXTDIRS)" -mcpponly  \
   -shh:uc:sc greet.idl

server_manager.obj: server_manager.c
   $(CCC) /I$(INCS) $(LOCAL_CCFLAGS) $?
```

```
clean:
   -$(REMOVE) GreetServer1.exe GreetClient.exe \
   Server_CORBA.obj client_s2.obj server_manager.obj client_CORBA.obj \
   greet_I.obj greet_C.obj greet_S.obj \
   greet_C.cpp greet_S.cpp \
   greet.hh
```

CORBA IDL file

Description: CORBA interface definition
File name: greet.idl

```
interface Greet {
  void greet_CORBA(in string client_greeting, out string server_reply);
};
```

Header file for C wrapper functions

Description: Header file containing the function prototype of all the C wrapper functions
File name: CWrapper.h

```
#ifdef __cplusplus
extern "C"{
#endif
  int initialize( int argc, char* argv[] );
  void finalize();
  void greet_CORBA( char* client_greeting, char** reply );
  void greet_mirror( const char * client_greeting, char * server_reply );
#ifdef __cplusplus
}
#endif

# define STR_SZ 20
```

C++ client for the revised application

Description: CORBA C++ client code
File name: client_CORBA.cpp

```
#ifndef lint
static const char *sccsid = "@(#) 1.0 GreetClient.cpp, 11/5/03";
#endif

#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>
#include <malloc.h>

#include <CosNaming.hh>

/* Local includes follow */
#include "greet.hh"
#include "CWrapper.h"

// Global declarations:
static ::CORBA::ORB_ptr op;
Greet_var greet_Impl, liptr = NULL;

// This function deallocates resources used throughout the program.

void release_resources( ::CORBA::ORB_ptr op ) {
  // Deallocate the various resources we have allocated.
  ::CORBA::release( op );
}

// This function initializes the ORB.

int perform_initialization( int argc, char *argv[]) {
  // Initialize the ORB.
  op = ::CORBA::ORB_init(argc, argv, "DSOM");
  cout << "Initialized ORB" << endl;

  return( 0 );
}

// This function accesses the Name Service and then gets
// the root naming context, which it returns;
// the Greet context.

::CosNaming::NamingContext_ptr get_naming_context() {
  ::CosNaming::NamingContext_ptr rootNameContext = NULL;
  ::CORBA::Object_ptr objPtr;

  // Get access to the Naming Service.
  try {
    objPtr = op->resolve_initial_references( "NameService" );
  }
  catch( ::CORBA::ORB::InvalidName e ) {
    cerr << "ERROR: resolve_initial_references threw InvalidName" << endl;
    release_resources( op );
    return( NULL );
  }
  catch( ::CORBA::SystemException e ) {
    cerr << "ERROR: resolve_initial_references threw SystemException"
```

```
      << endl;
    release_resources( op );
    return( NULL );
  }
  if ( objPtr == NULL ) {
    cerr << "ERROR: resolve_initial_references returned NULL" << endl;
    release_resources( op );
    return( NULL );
  }
  else
    cout << "resolve_initial_references returned = " << objPtr << endl;

  // Get the root naming context.
  rootNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
  if ( ::CORBA::is_nil( rootNameContext ) ) {
    cerr << "ERROR: rootNameContext narrowed to nil" << endl;
    release_resources( op );
    return( NULL );
  }
  else
    cout << "rootNameContext = " << rootNameContext << endl;
  // Release the temporary pointer.
  ::CORBA::release(objPtr );

  return( rootNameContext );
}

int initialize( int argc, char* argv[] ) {
  int rc;
  ::CORBA::Object_ptr objPtr;
  ::CosNaming::NamingContext_var rootNameContext = NULL;

  if ( ( rc = perform_initialization( argc, argv ) ) != 0 )
    return rc;

   cout << "Before getting naming context " << endl;

  // Get the root naming context.
  rootNameContext = get_naming_context();
  if ( ::CORBA::is_nil( rootNameContext ) )
    return -1;

  // Find the Greet_Impl created by the server. Look up the
  // object using the complex name of domain.GreetContext.GreetObject1,
  // which is its full name from the root naming context, as created
  // by GreetServer.

  cout << "Before CosNaming" << endl ;
  try {
```

```
    // Create a new ::CosNaming::Name to pass to resolve().
    // Construct it as the full three-part complex name starting at legacyRoot.
    ::CosNaming::Name *greetName = new ::CosNaming::Name;
    greetName->length( 3 );
    (*greetName)[0].id = ::CORBA::string_dup( "legacyRoot" );
    (*greetName)[0].kind = ::CORBA::string_dup( "" );
    (*greetName)[1].id = ::CORBA::string_dup( "GreetContext" );
    (*greetName)[1].kind = ::CORBA::string_dup( "" );
    (*greetName)[2].id = ::CORBA::string_dup( "GreetObject1" );
    (*greetName)[2].kind = ::CORBA::string_dup( "" );
    ::CORBA::Object_ptr objPtr = rootNameContext->resolve( *greetName );
    delete greetName;
    liptr = Greet::_narrow( objPtr);
    cout << "After narrow, liptr = " << liptr << endl;
  }
  catch( ::CosNaming::NamingContext::NotFound e ) {
    cerr << "ERROR: resolve threw NotFound" << endl;
    release_resources( op );
    return 0;
  }
  catch( ::CosNaming::NamingContext::CannotProceed e ) {
    cerr << "ERROR: resolve threw CannotProceed" << endl;
    release_resources( op );
    return 0;
  }
  catch( ::CosNaming::NamingContext::InvalidName e ) {
    cerr << "ERROR: resolve threw InvalidName" << endl;
    release_resources( op );
    return 0;
  }
  catch( ::CORBA::SystemException e ) {
    cerr << "ERROR: resolve rootNameContext threw SystemException"
      << endl;
    release_resources( op );
    return( 0 );
  }

  return 0;
}

void finalize() {
    release_resources( op );
}

void greet_CORBA( char* client_greeting, char** reply ) {
   char* server_reply;
    liptr->greet_CORBA( client_greeting, server_reply );
   *reply=server_reply;
}
```

C client for the revised application

Description: The revised C client code
File name: client_s2.c

```
#include <stdio.h>
#include "CWrapper.h"

int main ( int ac, char* av[] ) {
   int i, MAX_PASS;
   char  *name_to_greet, *reply;

   if ( ac !=3 ){
     fprintf( stderr, "Usage: %s message passes \n",av[0] );
     exit( 1 );
   }

   /* A call to the CORBA C++ client to initialize the client,
      get naming context and to access the servant object */
   initialize( ac,av );
   printf( "after initialize \n" );

   name_to_greet = av[1];
   MAX_PASS = atoi(av[2]);
   for ( i=1; i<=MAX_PASS; i++ ){
   /* Access the servant object method.
      In the DCE applciation, the function call was :
      greet_RPC(name_to_greet,reply); */
     greet_CORBA(name_to_greet, &reply);
     printf( "The Greet Server said : %s\n", reply );
     fflush( stdout );
   }

   /* A call to the CORBA C++ client to deallocate the resources */
   finalize();
   return 0;
}
```

Header file for CORBA servant

Description: Header file for servant implementation class
File name: greet.ih

```
//-----------------------------------------------------------------------------
//
//  Generated from greet.idl
//  On Thursday, May 15, 2003 12:08:57 PM CDT
//  by IBM CORBA 2.3 (ih) C++ emitter 2.30
```

```
//
//----------------------------------------------------------------------------

#ifndef _greet_ih_included
#define _greet_ih_included
    //====================================================//
    // DEVELOPER_NOTE:                                    //
    // The classes and code emitted in this file are      //
    // provided to assist you in the development of your  //
    // implementation objects, valuetypes, and valuetype  //
    // factories.  It is your responsibility to provide   //
    // the method implementations (see the corresponding  //
    // <name>_I.cpp file).  There are alternative         //
    // ways to design the implementation classes.         //
    // The following code represents one design approach. //
    //                                                    //
    // Please search for DEVELOPER_NOTE in this file      //
    // to locate all of the places where you              //
    // may need to examine, add, or modify code.          //
    //====================================================//


#ifdef SOMCBNOLOCALINCLUDES
#else
#endif
#ifdef SOMCBNOLOCALINCLUDES
#include <greet.hh>
#else
#include "greet.hh"
#endif

class Greet_Impl : public virtual ::Greet_Skeleton {

   public:
     ::CORBA::Void  greet_CORBA( const char* client_greeting,
                                 ::CORBA::String_out server_reply );

/* The following declarations for the constructor and destructor have been
   added to this generated file */
   public:
     Greet_Impl();
     virtual ~Greet_Impl();
};

#endif /* _greet_ih_included */
```

Servant implemention

Description: Servant implementation code
File name: greet_I.cpp

```
//--------------------------------------------------------------------------------
//
//  Generated from greet.idl
//  On Thursday, May 15, 2003 12:08:57 PM CDT
//  by IBM CORBA 2.3 (ic) C++ emitter 2.30
//
//--------------------------------------------------------------------------------

    //====================================================//
    // DEVELOPER_NOTE:                                    //
    // The classes and code emitted in this file are      //
    // provided to assist you in the development of your  //
    // implementation objects, valuetypes, and valuetype  //
    // factories.  It is your responsibility to provide   //
    // the method implementations and ensure correctness  //
    // (see the corresponding <name>.ih file).  There are //
    // alternative ways to design the implementation      //
    // classes.  The following code represents one        //
    // design approach.                                   //
    //                                                    //
    // Please search for DEVELOPER_NOTE in this file      //
    // to locate all of the places where you              //
    // may need to examine, add, or modify code.          //
    //====================================================//


#ifdef SOMCBNOLOCALINCLUDES
#include <greet.ih>
#else
#include "greet.ih"
#endif

#include "CWrapper.h"

::CORBA::Void  Greet_Impl::greet_CORBA( const char* client_greeting,
                                        ::CORBA::String_out server_reply ) {
  // DEVELOPER_NOTE: Provide method implementation
    char reply[ STR_SZ ];
    greet_mirror( client_greeting, reply );
    server_reply=::CORBA::string_dup( reply );
}

// Constructor for class Greet_Impl.
Greet_Impl::Greet_Impl() { }
```

```
// Destructor for class Greet_Impl.
Greet_Impl::~Greet_Impl() { }
```

Revised application server

Description: CORBA C++ server code
File name: server_CORBA.cpp

```
#ifndef lint
static const char *sccsid = "@(#) 1.0 GreetServer.cpp, 11/5/03 ";
#endif

#include <locale.h>
#include <CosNaming.hh>
#include <greet.ih>
#include <WSServerShutdown.h>

#if defined(minor)
#undef minor
#endif

// Global declarations:
static ::CORBA::ORB_ptr op;
static ::CORBA::BOA_ptr bp;
::CORBA::ImplementationDef_ptr imp ;
Greet_var greet_Impl;

// This function deallocates resources used throughout the program.

void release_resources( ::CORBA::BOA_ptr bp, ::CORBA::ImplementationDef_ptr
                   imp, ::CORBA::ORB_ptr op ) {
  // Deallocate the various resources we have allocated.
  bp->deactivate_impl( imp );
  ::CORBA::release( bp );
  ::CORBA::release( op );
  ::CORBA::release( imp );
}

// This function performs general initializtion, including retrieval
// of the appropriate ImplementationDef and initalization of the ORB and BOA.

int perform_initialization( int argc, char *argv[] ) {
  // Initialize the server's Implementation Repository.
  ::CORBA::ImplRepository_ptr implrep = new ::CORBA::ImplRepository();
  // Retrieve the appropriate ImplementationDef by using the server alias.
  try {
    imp = implrep->find_impldef_by_alias( argv[1] );
```

```
  }
  catch( ::CORBA::SystemException &ex ) {
    cerr << "ERROR: SystemException minor = " << ex.minor() <<
      " and id = " << ex.id();
    cerr << " was received when calling find_impldef_by_alias()" << endl;
    return( -1 );
  }
  cout << "Retrieved ImplementationDef" << endl;

  // Initialize the ORB.
  op = ::CORBA::ORB_init(argc, argv, "DSOM");

  // Initialize the BOA.
  try {
    bp = op->BOA_init(argc, argv, "DSOM_BOA");
  }
  catch( ::CORBA::SystemException &ex ) {
    cerr << "ERROR: SystemException minor = " << ex.minor() <<
      " and id = " << ex.id();
    cerr << " was received when calling BOA_init()" << endl;
    return( -1 );
  }
  cout << "Initialized ORB" << endl;

  // Initialize this application as a server, allow it to accept
  // incoming request messages, but do not register it with the somorbd
  // daemon (because this is a lightweight server of transient objects).
  try {
    bp->impl_is_ready( imp, 0 );
  }
  catch( ::CORBA::SystemException &ex ) {
    cerr << "ERROR: SystemException minor = " << ex.minor() <<
      " and id = " << ex.id();
    cerr << " was received when searching for the server" << endl;
    return( -1 );
  }
  cout << "Finished initialization of implementation" << endl;

  return( 0 );
}

// This function accesses the Name Service and then gets or creates
// the desired naming contexts. It returns the naming context for
// the Greet context.

::CosNaming::NamingContext_ptr get_naming_context() {
  ::CosNaming::NamingContext_var rootNameContext = NULL;
  ::CosNaming::NamingContext_var domainNameContext = NULL;
  ::CosNaming::NamingContext_ptr greetNameContext = NULL;
```

```
  ::CORBA::Object_ptr objPtr;

  // Get access to the Naming Service.
  try {
    objPtr = op->resolve_initial_references( "NameService" );
  }
  catch( ::CORBA::ORB::InvalidName e ) {
    cerr << "ERROR: resolve_initial_references threw InvalidName" << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  catch( ::CORBA::SystemException e ) {
    cerr << "ERROR: resolve_initial_references threw SystemException"
      << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  if ( objPtr == NULL ) {
    cerr << "ERROR: resolve_initial_references returned NULL" << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  else
    cout << "resolve_initial_references returned = " << objPtr << endl;

  // Get a root naming context.
  rootNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
  if ( ::CORBA::is_nil( rootNameContext ) ) {
    cerr << "ERROR: rootNameContext narrowed to nil" << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  else
    cout << "rootNameContext = " << rootNameContext << endl;
  // Release the temporary pointer.
  ::CORBA::release(objPtr);

  // Create a ::CosNaming::Name for legacyRoot, to get to the cell persistent
root.
  ::CosNaming::NameComponent nc;
  nc.kind = CORBA::string_dup("");
  nc.id = CORBA::string_dup("legacyRoot");
  ::CosNaming::Name_var name = new ::CosNaming::Name( 1, 1, &nc, 0 );
  // Get the legacyRoot naming context.
  try {
    objPtr = rootNameContext->resolve( name );
    cout << "objPtr from nameContext resolve = " << objPtr << endl;
  }
  catch( ::CosNaming::NamingContext::NotFound e ) {
```

```
    cerr << "ERROR: resolve threw NotFound" << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  catch( ::CosNaming::NamingContext::CannotProceed e ) {
    cerr << "ERROR: resolve threw CannotProceed" << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  catch( ::CosNaming::NamingContext::InvalidName e ) {
    cerr << "ERROR: resolve threw InvalidName" << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  catch( ::CORBA::SystemException e ) {
    cerr << "ERROR: resolve_initial_references threw SystemException"
      << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  cout << "Resolved legacyRoot in root name context" << endl;

  domainNameContext = ::CosNaming::NamingContext::_narrow(objPtr);
  if ( ::CORBA::is_nil( domainNameContext ) ) {
    cerr << "ERROR: domainNameContext narrowed to null" << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  cout << "domainNameContext = " << domainNameContext << endl;
  // Release the temporary pointer.
  ::CORBA::release( objPtr );

  // Get a new Greet naming context for our objects.
  ::CosNaming::NameComponent nc2;
  nc2.kind = CORBA::string_dup("");
  nc2.id = CORBA::string_dup("GreetContext");
  ::CosNaming::Name_var name2 = new ::CosNaming::Name( 1, 1, &nc2, 0 );
  try {
    greetNameContext = domainNameContext->bind_new_context( name2 );
    cout << "bind_new_context, greetNameContext = " << greetNameContext <<
endl;
  }
  catch( ::CosNaming::NamingContext::NotFound e ) {
    cerr << "ERROR: bind_new_context threw NotFound" << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  catch( ::CosNaming::NamingContext::CannotProceed e ) {
    cerr << "ERROR: bind_new_context threw CannotProceed" << endl;
```

```
    release_resources( bp, imp, op );
    return( NULL );
  }
  catch( ::CosNaming::NamingContext::InvalidName e ) {
    cerr << "ERROR: bind_new_context threw InvalidName" << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  catch( ::CORBA::SystemException &ex ) {
    cerr << "ERROR: SystemException minor = " << ex.minor() <<
      " and id = " << ex.id();
    cerr << " was received when trying to bind_new_context" << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }
  catch( ::CosNaming::NamingContext::AlreadyBound e ) {
    cerr << "ERROR: bind_new_context threw AlreadyBound" << endl;
    cout << "Trying to resolve the context" << endl;
    try {
      ::CosNaming::Name *greetName = new ::CosNaming::Name;
      greetName->length( 1 );
      (*greetName)[0].id = ::CORBA::string_dup( "GreetContext" );
      (*greetName)[0].kind = ::CORBA::string_dup( "" );
      ::CORBA::Object_ptr objPtr = domainNameContext->resolve( *greetName );
   cout << "Before greetNameContext = " << greetNameContext << endl;
      greetNameContext = ::CosNaming::NamingContext::_narrow( objPtr );
   cout << "AFter greetNameContext = " << greetNameContext << endl;
      delete greetName;
    }
    catch( ::CosNaming::NamingContext::NotFound e ) {
      cerr << "ERROR: resolve threw NotFound" << endl;
      release_resources( bp, imp, op );
      return( NULL );
    }
    catch( ::CosNaming::NamingContext::CannotProceed e ) {
      cerr << "ERROR: resolve threw CannotProceed" << endl;
      release_resources( bp, imp, op );
      return( NULL );
    }
    catch( ::CosNaming::NamingContext::InvalidName e ) {
      cerr << "ERROR: resolve threw InvalidName" << endl;
      release_resources( bp, imp, op );
      return( NULL );
    }
    catch( ::CosNaming::NamingContext::AlreadyBound e ) {
      cerr << "ERROR: resolve threw AlreadyBound" << endl;
      release_resources( bp, imp, op );
      return( NULL );
    }
```

```
  }
  catch( ... ) {
    cerr << "Unknow Exception thrown..." << endl;
    release_resources( bp, imp, op );
    return( NULL );
  }

  return( greetNameContext );
}

int create_and_bind( ::CosNaming::Name *nc, ::CosNaming::NamingContext_var
                     greetNameContext ) {
  // Create a Greet object and a "stringified" IOR version of it.
  greet_Impl = new Greet_Impl();

  // Bind the object to this name in the greet naming context.
  try {
    greetNameContext->rebind( *nc, greet_Impl );
    cout << "bind of greetNameContext succeeded" << endl;
  }
  catch( ::CosNaming::NamingContext::NotFound e ) {
    cerr << "ERROR: bind threw NotFound" << endl;
    release_resources( bp, imp, op );
    return( -1 );
  }
  catch( ::CosNaming::NamingContext::CannotProceed e ) {
    cerr << "ERROR: bind threw CannotProceed" << endl;
    release_resources( bp, imp, op );
    return( -1 );
  }
  catch( ::CosNaming::NamingContext::InvalidName e ) {
    cerr << "ERROR: bind threw InvalidName" << endl;
    release_resources( bp, imp, op );
    return( -1 );
  }
  catch( ::CosNaming::NamingContext::AlreadyBound e ) {
    cerr << "ERROR: bind threw AlreadyBound" << endl;
    release_resources( bp, imp, op );
    return( -1 );
  }
  catch( ::CORBA::SystemException &ex ) {
    cerr << "ERROR: SystemException minor = " << ex.minor() <<
      " and id = " << ex.id();
    cerr << " was received when trying to bind greet naming context" << endl;
    release_resources( bp, imp, op );
    return( -1 );
  }
  return( 0 );
}
```

```
int main( int argc, char *argv[] ) {
  ::CORBA::Object_ptr objPtr;
  ::CORBA::Status stat;
  int rc = 0;

  setlocale(LC_ALL, "");

  // Validate the input parameters.
  if ( argc != 2 ) {
    cerr << "Usage: GreetServer <server_alias>" << endl;
    exit( -1 );
  }

  if ( ( rc = perform_initialization( argc, argv ) ) != 0 )
    exit( rc );

  // Get the various naming contexts.
  ::CosNaming::NamingContext_var greetNameContext = NULL;
  greetNameContext = get_naming_context();
  if ( ::CORBA::is_nil( greetNameContext ) )
    exit( -1 );

  // Get a new ::CosNaming::Name for our Greet_Impl object.
  // This is done here rather than in create_and_bind() so that the
  // name can be reused later, when terminating the server.
  ::CosNaming::Name *nc = new ::CosNaming::Name;
  nc->length( 1 );
  (*nc)[0].id = ::CORBA::string_dup( "GreetObject1" );
  (*nc)[0].kind = ::CORBA::string_dup( "" );

  // Create a new Greet_Impl object and bind it to the greetNameContext.
  if ( ( rc = create_and_bind( nc, greetNameContext ) ) != 0 )
    exit( -1 );

  // Create a WSServerShutdown object that can break the server out of the
  // method execute_request_loop() when we are ready to terminate
  // the server. The WSStopServer command will make the subsequent
  // invocation of execute_request_loop() return to the server.
  WSServerShutdown *shutdownObj = new WSServerShutdown( argv[1], bp );
  cout << "Created WSServerShutdown object" << endl;

  cout << endl;
  cout << "server listening...." << endl << endl;
  cout.flush();

  // Go into an infinite loop, servicing ORB requests as they are
  // received. execute_request_loop() will return when an external command,
  // WSStopServer, is executed.
```

```
  stat = bp->execute_request_loop( ::CORBA::BOA::SOMD_WAIT );

  cout << "execute_request_loop has returned!" << endl;

  // Terminate the server.

  // Unbind the Greet object from the greet naming context.
  cout << "Unbinding the Greet object" << endl;
  try {
    greetNameContext->unbind( *nc );
  }
  catch( ::CORBA::SystemException &ex ) {
    cerr << "ERROR: SystemException minor = " << ex.minor() <<
      " and id = " << ex.id();
    cerr << " was received when calling unbind()" << endl;
  }
  // Remove the greet naming context.
  try {
    greetNameContext->destroy();
  }
  catch( ::CosNaming::NamingContext::NotEmpty e ) {
    cerr << "ERROR: destroy threw NotEmpty" << endl;
  }

  release_resources( bp, imp, op );
  delete shutdownObj;
  delete nc;

  cout << "Exiting GreetServer..." << endl;
  cout.flush();
  return 0;
}
```

Revised application logic

Description: The C code for the application logic part revised to remove DCE dependencies

File name: server_manager.c

```
#include <stdio.h>
#include "CWrapper.h"

void greet_mirror( const char * client_greeting, char * server_reply ) {
   char tmp[STR_SZ];
   int i,msglen;

   msglen = strlen( client_greeting );
   printf( "The client says: %s\n", client_greeting );
```

```
   for ( i=0;i<msglen;i++ ) {
     tmp[i] = client_greeting[msglen-i-1];
   }
   tmp[i] = '\0';
   printf( "This I turn around into %s\n",tmp );
   fflush( stdout );
   strncpy( server_reply,tmp,msglen );
}
```

Properties file for client

Description: Properties file for the CORBA C++ client
File name: WSClient.props

```
# WebSphere Application Server Enterprise 5.0 props file
#
# The setting for com.ibm.CORBA.TCPIP.lsdport is the number
# of the port used by the location service daemon.
# (If not specified, ASV's default value for lsdPort is 9000.)
#
com.ibm.CORBA.TCPIP.lsdPort=9000

# The setting for com.ibm.CORBA.bootstrapHostName is the full name
# of the machine on which the name server (i.e., ASV) is running.
#
com.ibm.CORBA.bootstrapHostName=9.3.4.230

# The setting for com.ibm.CORBA.bootstrapPort is the number of the port
# that the name server uses to communicate with clients and servers.
# (If not specified, ASV's default value for bootstrapPort is 2809.)
#
com.ibm.CORBA.bootstrapPort=2809
```

Properties file for server

Description: Properties file for the CORBA C++ server
File name: WSServer.props

```
# WebSphere Application Server Enterprise 5.0 props file
#
# The setting for com.ibm.CORBA.hostName is the full name of the
# system that is running the server code.
#
com.ibm.CORBA.hostName=tivdce3.itsc.austin.ibm.com
# The setting for com.ibm.CORBA.TCPIP.lsdport is the number
# of the port used by the location service daemon.
# (If not specified, ASV's default value for lsdPort is 9000.)
```

```
#
com.ibm.CORBA.TCPIP.lsdPort=9000
com.ibm.CORBA.serverListenPort=1111

# The setting for com.ibm.CORBA.bootstrapHostName is the full name
# of the machine on which the name server (i.e., ASV) is running.
#
com.ibm.CORBA.bootstrapHostName=tivdce3.itsc.austin.ibm.com

# The setting for com.ibm.CORBA.bootstrapPort is the number of the port
# that the name server uses to communicate with clients and servers.
# (If not specified, ASV's default value for bootstrapPort is 2809.)
#
com.ibm.CORBA.bootstrapPort=2809
```

C

Scenario 3: Source code listings

This appendix lists the complete source code as that was explained in pieces in Chapter 10, “Scenario 3: Secure RPC application #1” on page 195. Specifically, this appendix contains:

- The source code of the client and server programs with DCE dependencies
- The source code of the revised client and server programs without DCE dependencies
- The makefiles used for these programs
- Additional source code modules for authorization
- Header files
- Application configuration files

Refer to Appendix E, “Additional material” on page 417 for instructions for downloading the source code samples included in this appendix.

© Copyright IBM Corp. 2003. All rights reserved.

Application with DCE dependencies

The files are:

- Makefile contains information about how the application is compiled and linked on the AIX operating system.
- Directory.mak contains information about how the application is compiled and linked on the Windows operating system.
- Directory.idl contains the interface definition shared by client and server programs.
- client_s3.c contains the client code. The application client looks up a binding to the application server via DCE CDS and extracts the server's principal name from that binding. It then tests whether the server is a member of the security group it requests. If the test is OK, the client annotates the binding handle with its security information and starts sending application data to the server.
- server_s3.c contains the server code. First, the server performs a login to DCE and creates threads in which its credentials and key are periodically renewed. Then the server creates and registers its binding information with DCE CDS Naming Service and DCE RPC endpoint mapper. If this is OK, the server starts listening for client calls.
- directory_mgr.c implements the DCE authorization part. For every client request, the server enforces the right level of protection and tests for correct group membership of the calling principal. If access is allowed, the client requests are forwarded to the actual business logic in directory_impl.c
- directory_impl.h provides forward declarations of the business services of directory_impl.c
- directory_impl.c contains the application code of the server application. The application consists of three services that return employee information.
- checks_status.h provides common error-handling code.

Makefile for the AIX platform

Description: Makefile for AIX application client and server
File name: Makefile

```
SOURCE = server_s3.c client_s3.c directory_mgr.c directory_impl.c
SOBJS = server_s3.o directory_mgr.o directory_impl.o Directory_sstub.o
COBJS = client_s3.o Directory_cstub.o
IDL = /usr/bin/idl
IDL_INCDIRS = -I..
IDL_FLAGS = ${IDL_INCDIRS} -keep c_source
```

```
INCDIRS = -I/usr/include/dce -I.
LIBDIRS = -L/usr/lib -L.
LIBS = ${LIBDIRS} -ldce -ldcepthreads -lpthreads
CDEFS = -Dunix -qlanglvl=extended -D_ALL_SOURCE -D _DCE_PTHREADS
CFLAGS  = -g -w ${CDEFS} ${INCDIRS}
CC = cc
.c.o:
   ${CC} ${CFLAGS} -c $<

all:server_s3 client_s3

server_s3.o:  server_s3.c Directory.h
   ${CC} ${CFLAGS} -c server_s3.c

client_s3.o:  client_s3.c Directory.h
   ${CC} ${CFLAGS} -c client_s3.c

directory_impl.o: directory_impl.c Directory.h
   ${CC} ${CFLAGS} -c directory_impl.c

directory_mgr.o: directory_mgr.c Directory.h
   ${CC} ${CFLAGS} -c directory_mgr.c

Directory_sstub.o: Directory_sstub.c Directory.h
   ${CC} ${CFLAGS} -c Directory_sstub.c

Directory_cstub.o: Directory_cstub.c Directory.h
   ${CC} ${CFLAGS} -c Directory_cstub.c

Directory_cstub.c Directory_sstub.c Directory.h: Directory.idl
   ${IDL} Directory.idl ${IDL_FLAGS}

server_s3: ${SOBJS}
   ${CC} -o server_s3 ${SOBJS} ${LIBS}

client_s3: ${COBJS}
   ${CC} -o client_s3 ${COBJS} ${LIBS}

clean:
   rm -f Directory.h Directory_sstub.c Directory_cstub.c ${SOBJS} ${COBJS}

rmtarget:
   rm -f server_s3 client_s3

clobber: clean rmtarget
```

Makefile for the Windows platform

Description: Makefile for Windows application client and server
File name: Directory.mak

```
!include <NTWIN32.MAK>

!if "$(CPU)" == "ALPHA"
TARGET=alpha
SWITCHES=-DALPHA
!else
TARGET=intel
SWITCHES=-D_X86=1 -DM_I86
!endif

_LIBS_ = \
   libdce.lib \
   pthreads.lib \
   $(conlibsdll)

IDL = idl -keep c_source -v

CFLAGS= -I. $(cflags:-W3=) $(cvarsdll) $(SWITCHES) -Zi -Od -Gz -nologo

LINK = link $(ldebug) $(conlflags)

all: client_s3.exe server_s3.exe

clean :
   -del *.obj
   -del *.exe
   -del Directory_cstub.c
   -del Directory_sstub.c
   -del Directory.h

client_s3.exe : client_s3.obj Directory_cstub.obj directory_impl.obj
   $(LINK) /out:$@ $** $(_LIBS_)

server_s3.exe : server_s3.obj directory_mgr.obj directory_impl.obj
Directory_sstub.obj
   $(LINK) /out:$@ $** $(_LIBS_)

directory_cstub.obj : Directory_cstub.c Directory.h
   $(CC) $(CFLAGS) Directory_cstub.c

server_s3.obj : server_s3.c Directory.h
   $(CC) $(CFLAGS) server_s3.c

directory_mgr.obj : Directory_mgr.c Directory.h
```

```
   $(CC) $(CFLAGS) Directory_mgr.c

directory_impl.obj : Directory_impl.c directory_impl.h Directory.h
   $(CC) $(CFLAGS) Directory_impl.c

client_s3.obj : client_s3.c Directory.h
   $(CC) $(CFLAGS) client_s3.c

Directory_sstub.obj : Directory_sstub.c Directory.h
   $(CC) $(CFLAGS) Directory_sstub.c

Directory_cstub.c Directory_sstub.c Directory.h : Directory.idl
   $(IDL) -client stub -server stub  -cstub Directory_cstub.c \
   -sstub Directory_sstub.c Directory.idl
```

IDL source

Description: Interface definition
File name: Directory.idl

```
[
   uuid(e2f98da0-8a19-11d7-8a68-00609437fb07),
   version(1.0),
   pointer_default(ptr)
]
interface Directory
{
 typedef [string] char *dept_name_t;
    void get_dept(
       [in] handle_t handle,
       [in,string] char *emp_name,
       [out] dept_name_t *dept_name
    );
    void get_grade(
       [in] handle_t handle,
       [in, string] char *emp_name,
       [out]long *grade
    );
    void get_salary(
       [in] handle_t handle,
       [in, string] char *emp_name,
       [out] double *salary
    );
}
```

DCE dependent application client

Description: DCE application client program
File name: client_s3.c

```
#include <stdio.h>
#include "Directory.h"
#include "check_status.h"
#include <string.h>
#include <dce/binding.h>
#include <dce/pgo.h>
#include <dce/secidmap.h>
#define SERVER_CDS "/.:/directory_server"
#define SERVER_GROUP "directory_server"
rpc_binding_handle_t binding_handle;
void show_usage(int argc, char *argv[]) {
   printf("Usage is: %s <-function {0|1|2}> <-employee emp_name>",argv[0]);
   exit(1);
}
void execute_query(int argc, char **argv) {
   int i=0;
   int query_type=0;
   enum query_type {DEPT,GRADE,SALARY};
   char *emp_name="";
   long grade;
   double salary;
   char *dept_name="";
   for(i=0;i<argc;i++) {
      if(strcmp(argv[i],"-function")==0) {
         if(i<argc-1) {
            query_type=atoi(argv[i+1]);
            i++;
         }
      }
      if(strcmp(argv[i],"-employee")==0) {
         if(i<argc-1) {
            emp_name=argv[i+1];
            i++;
         }
      }
      if(strcmp(argv[i],"-?")==0) show_usage(argc,argv);
   }
   switch(query_type) {
      case DEPT: {
         get_dept (binding_handle, emp_name, &dept_name);
         printf ("Department for %s is: %s\n",
            strcmp(emp_name,"")!=0?emp_name:"current user",
            dept_name
         );
```

```
         break;
      }
      case GRADE: {
         get_grade (binding_handle, emp_name, &grade);
         if(grade<0) {
            printf (
               "Grade for %s is: unknown\n",
               strcmp(emp_name,"")!=0?emp_name:"current user"
            );
         }
         else {
            printf (
               "Grade for %s is: %i\n",
               strcmp(emp_name,"")!=0?emp_name:"current user",
               grade
            );
         }
         break;
      }
      case SALARY:{
         get_salary(binding_handle, emp_name, &salary);
         if(salary<0) {
            printf (
               "Salary for %s is: unknown\n",
               strcmp(emp_name,"")!=0?emp_name:"current user"
            );
         }
         else {
            printf (
               "Salary for %s is: %f.2\n",
               strcmp(emp_name,"")!=0?emp_name:"current user",
               salary
            );
         }
         break;
      }
      default: {
         printf("Invalid query type %d\n",query_type);
         show_usage(argc,argv);
         break;
      }
   }
}
void main(int argc,char *argv[]) {
   unsigned32 status;
   rpc_ns_handle_t import_context;
   sec_rgy_handle_t rgy_handle;
   sec_rgy_name_t princ_name;
   unsigned_char_t *server_princ_name;
```

```
boolean32 is_member;
/*
 * Get server binding from the name space.
 */
rpc_ns_binding_import_begin(
   rpc_c_ns_syntax_dce,
   SERVER_CDS,
   Directory_v1_0_c_ifspec,
   NULL,
   &import_context,
   &status
);
CHECK_STATUS (status, "Import begin failed", ABORT);
rpc_ns_binding_import_next(import_context,&binding_handle, &status);
CHECK_STATUS (status, "Import next failed", ABORT);
rpc_ns_binding_import_done(&import_context, &status);
CHECK_STATUS (status, "Import done failed", ABORT);
rpc_ep_resolve_binding (
   binding_handle,
   Directory_v1_0_c_ifspec,
   &status
);
CHECK_STATUS (status, "resolve_binding failed", ABORT);
/*
 * Determine the server's principal name.
 */
rpc_mgmt_inq_server_princ_name (
   binding_handle,
   rpc_c_authn_dce_secret,
   &server_princ_name,
   &status
);
CHECK_STATUS (status, "inq_princ_name failed", ABORT);
/*
 * Open a registry site for query
 */
sec_rgy_site_open_query(NULL, &rgy_handle, &status);
CHECK_STATUS (status, "rgy_site_open failed", ABORT);
/*
 * Ask the Security registry to translate the global principal name into
 * a simple principal name.
 */
sec_id_parse_name (
   rgy_handle,
   server_princ_name,
   NULL,
   NULL,
   princ_name,
   NULL,
```

```
      &status
   );
   CHECK_STATUS (status, "sec_id_parse_name failed", ABORT);
   /*
    * Find out if the principal name that is returned by the server
    * is a member of the directory_server group.
    */
   is_member = sec_rgy_pgo_is_member (
      rgy_handle,
      sec_rgy_domain_group,
      SERVER_GROUP,
      princ_name,
      &status
   );
   CHECK_STATUS (status, "is_member failed", ABORT);
   /*
    * We are done with the registry; we can release the rgy_handle now.
   */
   sec_rgy_site_close(rgy_handle, &status);
   CHECK_STATUS (status, "rgy_site_close failed", ABORT);
   if (! is_member ) {
         printf ("Found an invalid directory server\n");
         exit(1);
   }
   /*
    * Annotate binding handle for authentication.
    */
   rpc_binding_set_auth_info(
      binding_handle,
      server_princ_name,
      rpc_c_protect_level_pkt_privacy,
      rpc_c_authn_dce_secret, NULL,
      rpc_c_authz_dce, &status
   );
   CHECK_STATUS (status, "binding_set_auth_info failed", ABORT);
   execute_query(argc,argv);
   exit (0);
}
```

DCE dependent application server

Description: DCE application server program
File name: server_s3.c

```
#include <stdio.h>
#include <pthread.h>
#include "Directory.h"
#include "check_status.h"
```

```
#include <dce/keymgmt.h>
#include <dce/sec_login.h>
#define SERVER_PRINCIPAL_NAME "directory_server"
#define KEYTAB "directory_server_tab"
#define SERVER_CDS "/.:/directory_server"
#define ABORT 1
#define MINUTE 60
void establish_identity();

void main(int argc,char *argv[]) {
   rpc_binding_vector_t *bind_vector_p;
   unsigned32 status;
   /*
    * Register interface with rpc runtime.
    */
   rpc_server_register_if(Directory_v1_0_s_ifspec, NULL,NULL, &status);
   CHECK_STATUS (status,"unable to register i/f",ABORT);
   /*
    * We want to use all supported protocol sequences.
    */
   rpc_server_use_all_protseqs(rpc_c_protseq_max_reqs_default, &status);
   CHECK_STATUS (status,"use_all_protseqs failed",ABORT);
   /*
    * Establish this server's identity by setting up the appropriate login
context.
    */
   establish_identity (&status);
   CHECK_STATUS (status,"Cannot establish server identity",ABORT);
   /*
    * Register authentication info with RPC.
    */
   rpc_server_register_auth_info(
      SERVER_PRINCIPAL_NAME,
       rpc_c_authn_dce_secret,
      NULL,
       KEYTAB,
      &status
   );
   CHECK_STATUS (status,"server_register_auth_info failed",ABORT);
   /*
    * Continue with the normal initialization sequence. Get our bindings...
    */
   rpc_server_inq_bindings(&bind_vector_p, &status);
   CHECK_STATUS (status,"server_inq_bindings failed",ABORT);
   /*
    * Register binding information with the endpoint map.
    */
   rpc_ep_register(
      Directory_v1_0_s_ifspec,
```

```
      bind_vector_p,
      NULL,
      (unsigned_char_t *)"Directory server, version 1.0",
      &status
   );
   CHECK_STATUS (status,"ep_register failed",ABORT);
   /*
    * Export binding information into the namespace.
    */
   rpc_ns_binding_export(
      rpc_c_ns_syntax_dce,
      SERVER_CDS,
      Directory_v1_0_s_ifspec,
      bind_vector_p,
      NULL,
      &status
   );
   CHECK_STATUS (status,"export failed",ABORT);
   /*
    * Listen for remote calls.
   */
   printf("Server ready.\n");
   rpc_server_listen(rpc_c_listen_max_calls_default, &status);
   CHECK_STATUS (status,"server_listen failed",ABORT);
   /*
    * We don't expect to return from the listen loop.
    */
   printf("Unexpected return from rpc_server_listen\n");
   exit(1);
}
/*
 * Internal routine to establish this server as the
 * department_server principal.
 */
void establish_identity (error_status_t *o_status)
{
   sec_login_handle_t login_context;
      sec_login_auth_src_t auth_src;
      void *server_key;
      error_status_t status;
      boolean32 identity_valid;
      boolean32 reset_passwd;
      pthread_t refresh_login_context_thread;
      pthread_t key_mgmt_thread;
      void refresh_login_context_rtn ();
      void key_mgmt_rtn ();
      /*
       * Set up the network identity for this server principal.
       * The network credentials obtained are sealed and must be
```

```
     * unsealed with the server's secret key before they can
     * be used.
     */
    sec_login_setup_identity(
    SERVER_PRINCIPAL_NAME,
    sec_login_no_flags,
    &login_context,
    &status
);
    CHECK_STATUS (status,"unable to set up identity",ABORT);
/*
 * Retrieve the server's secret key from the private keytab file.
 */
 sec_key_mgmt_get_key(
    rpc_c_authn_dce_secret,
    KEYTAB,SERVER_PRINCIPAL_NAME,
    0,
    &server_key,
    &status
);
 CHECK_STATUS (status,"unable to retrive key",ABORT);
    /*
 * Unseal the network identity using the server's secret key.
 */
 identity_valid = sec_login_validate_identity(
    login_context,
    server_key,
    &reset_passwd,
    &auth_src,
    &status
);
    /*
 * Free the secret key as soon as we are done with it.
 */
sec_key_mgmt_free_key (server_key, &status);
CHECK_STATUS (status,"unable to free key",ABORT);
/*
 * Make sure that the server identity was validated by the network
 */
 if (identity_valid) {
    if (auth_src != sec_login_auth_src_network) {
          printf ("Server has no network credentials\n");
          exit (1);
       }
    /*
 * We make this login context the default for this process.
 */
    sec_login_set_context(login_context, &status);
    CHECK_STATUS (status,"unable to set login context",ABORT);
```

```
   /*
    * Start up a thread to refresh the login context when it expires.
    */
      if ((
         pthread_create (
            &refresh_login_context_thread,
            pthread_attr_default,
            (pthread_startroutine_t) refresh_login_context_rtn,
            (pthread_addr_t)login_context
         )
      ) == -1) exit (1);
   /*
    * Start up a thread to manage our secret key.
    */
      if ((
         pthread_create (
            &key_mgmt_thread,
            pthread_attr_default,
            (pthread_startroutine_t) key_mgmt_rtn,
            (pthread_addr_t)NULL
         )
      ) == -1) exit (1);
      *o_status=status;
    }
    else
    {
         error_status_t temp_status;
         CHECK_STATUS (status,"unable to validate network identity",0);
   /*
    * Reclaim the storage
    */
      sec_login_purge_context (&login_context, &temp_status);
      CHECK_STATUS (temp_status,"unable to purge login context",ABORT);
        *o_status = status;
   }
   return;
}
/*
 * A thread to periodically change the server's secret key.
 */
void key_mgmt_rtn ()
{
    error_status_t  status;
    while (1)
    {
        sec_key_mgmt_manage_key (
         rpc_c_authn_dce_secret,
         KEYTAB,
         SERVER_PRINCIPAL_NAME,
```

```
         &status
      );
        CHECK_STATUS (status,"key mgmt failure",ABORT);
    }
}
/*
 * A thread to periodically refresh the credentials contained
 * in a login context.
 */
void refresh_login_context_rtn (login_context)
sec_login_handle_t  login_context;
{
    signed32 expiration;
    signed32 delay_time;
    boolean32 reset_passwd;
    boolean32 identity_valid;
    void *server_key;
    sec_login_auth_src_t auth_src;
    error_status_t   status;
    while (1)
    {
      time_t current;
      time(&current);
      sec_login_get_expiration (login_context, &expiration, &status);
      if ((status != rpc_s_ok) && (status != sec_login_s_not_certified))
      {
            printf ("Cannot get login context expiration time\n");
            exit (1);
      }
   /*
    * Wait until shortly before the login context expires...
    */
      delay_time = expiration - current - (10*MINUTE);
      if (delay_time > 0) {
            struct timespec delay;
            delay.tv_sec = delay_time;
            delay.tv_nsec = 0;
            pthread_delay_np (&delay);
      }
      sec_login_refresh_identity (login_context, &status);
      CHECK_STATUS (status, "cannot refresh identity", ABORT);
   /*
    * Retrieve the server's secret key from the private keytab file.
    */
      sec_key_mgmt_get_key(
         rpc_c_authn_dce_secret,
         KEYTAB,SERVER_PRINCIPAL_NAME,
         0,
         &server_key,
```

```
         &status
      );
      CHECK_STATUS (status,"unable to retrive key",ABORT);
   /*
    * The refreshed login context still needs to be validated.
    */
      identity_valid = sec_login_validate_identity(
         login_context,
         server_key,
         &reset_passwd,
         &auth_src,
         &status);
   /*
     * Free the secret key as soon as we are done with it.
     */
        sec_key_mgmt_free_key (server_key, &status);
        CHECK_STATUS (status,"unable to free key",ABORT);
      if (! identity_valid)
      {
            error_status_t temp_status;
            sec_login_purge_context (&login_context, &temp_status);
            CHECK_STATUS (temp_status,"unable to purge login context",ABORT);
            CHECK_STATUS (status,"unable to validate network identity",ABORT);
      }
    }
}
```

Application server manager

Description: Application server manager and reference monitor
File name: directory_mgr.c

```
#include <stdio.h>
#include "Directory.h"
#include "check_status.h"
#include <dce/binding.h>
#include <dce/pgo.h>
#include <dce/secidmap.h>
#include <dce/id_base.h>
#include <string.h>
#include "directory_impl.h"
#define CONTINUE 0
#define UNAUTHENTICATED_USER "unauthenticated user"
int check_auth(
   rpc_binding_handle_t handle,
   char *type,
   char **client_name
) {
```

```
 sec_id_pac_t *pac;
 unsigned_char_t *server_principal_name;
 sec_rgy_name_t client_principal_name;
 unsigned32 protection_level;
 unsigned32 authn_svc;
 unsigned32 authz_svc;
 sec_rgy_handle_t rgy_handle;
 error_status_t status;
int is_valid=TRUE;
 /*
 * Check the authentication parameters that the
 * client selected for this call.
 */
 rpc_binding_inq_auth_client (
   handle,
   (rpc_authz_handle_t *) &pac,
   &server_principal_name,
   &protection_level,
   &authn_svc,
   &authz_svc,
   &status
);
 CHECK_STATUS (status, "inq_auth_client failed",CONTINUE);
 /*
 * Make sure that the caller has specified the required
 * level of protection, authentication, and authorization.
 */
 if (! (
   (protection_level == rpc_c_protect_level_pkt_privacy) &&
   (authn_svc == rpc_c_authn_dce_secret) &&
   (authz_svc == rpc_c_authz_dce)
) ) is_valid=FALSE;
 /*
 * Establish a binding to the registry interface of the
 * Security Server.
 */
 sec_rgy_site_open_query(NULL, &rgy_handle, &status);
 CHECK_STATUS (status, "rgy_site_open failed",CONTINUE);
 /*
 * Convert the UUID in the PAC into a name.
 */
 sec_rgy_pgo_id_to_name (
   rgy_handle,
   sec_rgy_domain_person,
   &(pac->principal.uuid),
   client_principal_name,
   &status
);
 CHECK_STATUS (status, "pgo_id_to_name failed",CONTINUE);
```

```
    /*
    * Check to see if the client principal is an employee
    */
   if(type!=NULL) {
      is_valid = sec_rgy_pgo_is_member (
         rgy_handle,
         sec_rgy_domain_group,
         type,
         client_principal_name,
         &status
      );
      CHECK_STATUS (status, "is_member failed", CONTINUE);
   }
    /*
    * We are done with the Security registry; free the handle now.
    */
    sec_rgy_site_close(rgy_handle,&status);
    CHECK_STATUS (status, "rgy_site_close failed",CONTINUE);
   *client_name=client_principal_name;
   return is_valid;
}

void IDL_STD_STDCALL get_grade(
   rpc_binding_handle_t handle,
   idl_char *emp_name,
   long *grade
) {
   char *name;
   if(check_auth(handle,"directory_employee",&name)==TRUE) {
      if(emp_name==NULL) emp_name=name;
      *grade=getGradeImpl(emp_name);
      return;
    }
   *grade = -2;
   return;
}
void  IDL_STD_STDCALL get_dept(
   handle_t handle,
   idl_char *emp_name,
   idl_char **dept
) {
   char *name;
   idl_char *the_dept;
   int size=0;
   if(check_auth(handle,NULL,&name)==TRUE) {
      if(emp_name==NULL) emp_name=name;
      the_dept=getDepartmentImpl(emp_name);
      size=strlen(the_dept);
      *dept=rpc_ss_allocate(size+1);
```

```
      strcpy(*dept,the_dept);
      return;
   }
   size=strlen(UNAUTHENTICATED_USER);
   *dept=rpc_ss_allocate(size+1);
   strcpy(*dept,UNAUTHENTICATED_USER);
   return;
}
void IDL_STD_STDCALL get_salary(
   rpc_binding_handle_t handle,
   idl_char *emp_name,
   double *salary
) {
   char *name;
   if(check_auth(handle,"directory_manager",&name)==TRUE) {
      if(emp_name==NULL) emp_name=name;
      *salary = getSalaryImpl(emp_name);
      return;
   }
      *salary = -2.0;
   return;
}
```

Application server logic header

Description: Header file for application server business logic
File name: directory_impl.h

```
#ifndef DIRECTORY_EXPORT
#define DIRECTORY_EXPORT
#endif
DIRECTORY_EXPORT char *getDepartmentImpl(char *name);
DIRECTORY_EXPORT double getSalaryImpl(char *name);
DIRECTORY_EXPORT long getGradeImpl(char *name);
```

Application server logic

Description: Application server business logic
File name: directory_impl.c

```
#include <stdio.h>
#include <string.h>
#include "directory_impl.h"
typedef struct {
      char *emp_name;
      char *dept_name;
   long grade_value;
```

```
   double salary_value;
} emp_entry_t;

emp_entry_t emp_table [] = {
   {"peter_morgan", "Accounting",1,24000.0},
      {"ruth_jones", "Marketing",2,38000.0},
      {"howard_stein","Finance",2,42000.0},
      {"fran_cooper", "Administration",1,27000.0},
      {"john_doe", "Training",1,18000.0},
   {NULL,NULL,-1,0}
};
DIRECTORY_EXPORT char *getDepartmentImpl(char *name) {
   emp_entry_t *emp_table_p=emp_table;
   int i=0;
      for (i=0;; i++) {
      if(emp_table_p->emp_name==NULL) return "unknown";
         if (! strcmp (name, emp_table_p->emp_name))
         return emp_table_p->dept_name;
      emp_table_p++;
   }
   return "unknown";
}
DIRECTORY_EXPORT long getGradeImpl(char *name) {
   emp_entry_t *emp_table_p=emp_table;
   int i=0;
      for (i=0;; i++) {
      if(emp_table_p->emp_name==NULL) return -1;
         if (! strcmp (name, emp_table_p->emp_name))
         return emp_table_p->grade_value;
      emp_table_p++;
   }
   return -1;
}
DIRECTORY_EXPORT double getSalaryImpl(char *name) {
   emp_entry_t *emp_table_p=emp_table;
   int i=0;
      for (i=0;; i++) {
      if(emp_table_p->emp_name==NULL) return -1.0;
         if (! strcmp (name, emp_table_p->emp_name))
         return emp_table_p->salary_value;
      emp_table_p++;
   }
   return -1.0;
}
```

Common error handling

Description: Common error handling for client and server.
File name: check_status.h

```
#include <stdio.h>
#include <dce/dce_error.h>
#define ABORT 1
#define CONTINUE 0
#define CHECK_STATUS(input_status,comment,type) \
{ \
   if(input_status != rpc_s_ok) { \
      dce_error_inq_text(input_status,error_string,&error_stat); \
      printf("%s %s\n", comment, error_string); \
      if(type==ABORT) exit(1); \
   } \
}

static int error_stat;
static unsigned char error_string[dce_c_error_string_len];
```

Revised application without DCE dependencies

The files are:

- JNIConnection.java implements the Connection interface, which calls methods in the back-end server by using JNI wrappers for the native C application server code.
- JNIConnectionFactory.java implements the ConnectionFactory interface for the end-user to use to create objects of type JNIConnection.
- JNIConnectionManager.java implements ConnectionManager interface and is responsible for pooling connections.
- JNIManagedConnection.java implements the ManagedConnection interface for the physical implemention of the connection.
- JNIConnectionMetaData.java returns information (meta data) about the JNIConnection class such as the product name, product version etc., of the back-end C application server.
- JNIManagedConnectionFactory.java implements the ManagedConnectionFactory interface to create objects of type JNIManagedConnection.
- JNIManagedConnectionMetaData.java returns information (meta data) about the JNIManagedConnection class such as the product name, product version etc., of the back-end C application server.

- JNIResourceAdapterMetaData.java returns information (meta data) about the resource adapter such as name of the resource adapter, vendor, transaction support, etc.
- DirectorySessionBeanImpl.java is the Enterprise Bean wrapper that encloses the back-end C application server code. It is a stateless session bean.
- DirectorySessionBean.java is the remote interface for the Enterprise Bean wrapper class DirectorySessionBeanImpl.
- DirectorySessionHome.java is the home interface for the Enterprise Bean wrapper class DirectorySessionBeanImpl.
- application.xml is the deployment descriptor for the whole enterprise application.
- ejb-jar.xml is the deployment descriptor for the Enterprise Bean wrapper.
- ra.xml is the deployment descriptor for the JCA resource adapter.
- directory_impl.c is the back-end C application server where all the application logic is handled.
- directory_jni_impl.c is the JNI wrapper around the back-end C application server.
- directory_impl.h is the header file for the C server program.
- corba_client.cpp is the CORBA C++ client that looks up the Enterprise Bean wrapper, which in turns talks to the JCA Connector.
- WSEJBClient.props is the client-side properties file for the CORBA client to specify the bootstrap host, bootstrap port, the bean to look up, and other run-time properties.

JNI connection

Description: JNI connection class which invokes the methods of the C application server
File name: JNIConnection.java

```
package com.ibm.redbook.scenario3.RA;

import java.io.PrintWriter;
import java.io.Serializable;
import java.util.Vector;

import javax.resource.NotSupportedException;
import javax.resource.ResourceException;
import javax.resource.cci.Connection;
import javax.resource.cci.ConnectionMetaData;
import javax.resource.cci.Interaction;
```

```
import javax.resource.cci.LocalTransaction;
import javax.resource.cci.ResultSetInfo;
import javax.security.auth.Subject;

public class JNIConnection implements Connection,Serializable {

   static public native String getDepartmentJNI(String emp_name);
   static public native int getGradeJNI(String emp_name);
   static public native double getSalaryJNI(String emp_name);
   static {
      System.loadLibrary("directory_jni");
   }

   private JNIManagedConnection manconn;
   public String getDepartment(String emp_name) {
   return getDepartmentJNI(emp_name);
   }
   public int getGrade(String emp_name) {
   return getGradeJNI(emp_name);
   }
   public double getSalary(String emp_name) {
      return getSalaryJNI(emp_name);
   }
   public JNIConnection(JNIManagedConnection mc) {
      manconn = mc;
   }
   public Interaction createInteraction() throws NotSupportedException {
      throw new NotSupportedException("JNIConnection");
   }
   public void close() {
   }
   public LocalTransaction getLocalTransaction()
      throws NotSupportedException {
      throw new NotSupportedException("JNIConnection");
   }
   public ConnectionMetaData getMetaData() throws NotSupportedException {
      throw new NotSupportedException("JNIConnection");
   }
   public ResultSetInfo getResultSetInfo() throws NotSupportedException {
      throw new NotSupportedException("JNIConnection");
   }
}
```

JNI connection factory class

Description: JNIConnection class to create objects of type JNIConnection
File name: JNIConnectionFactory.java

```
package com.ibm.redbook.scenario3.RA;

import java.io.Serializable;
import javax.naming.Reference;
import javax.resource.ResourceException;
import javax.resource.cci.Connection;
import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.ConnectionSpec;
import javax.resource.cci.RecordFactory;
import javax.resource.cci.ResourceAdapterMetaData;
import javax.resource.spi.ConnectionManager;
import javax.resource.spi.ConnectionRequestInfo;
import javax.resource.spi.ManagedConnectionFactory;

public class JNIConnectionFactory implements ConnectionFactory,Serializable {
   private Reference ref;
   private ConnectionManager conManager;
   private ManagedConnectionFactory mconFactory;
   public JNIConnectionFactory() {
   }
   public JNIConnectionFactory(
      ManagedConnectionFactory mcf,
      ConnectionManager conm) {
      conManager = conm;
      mconFactory = mcf;
   }
   public RecordFactory getRecordFactory() throws ResourceException {
      throw new ResourceException("cannot get a record factory");
   }
   public Connection getConnection() throws ResourceException {
      return (Connection) conManager.allocateConnection(mconFactory, null);
   }
   public Connection getConnection(
      javax.resource.cci.ConnectionSpec properties)
      throws ResourceException {
         System.out.println("CF get connection");
      return (Connection) conManager.allocateConnection(mconFactory, null);
   }
   public ResourceAdapterMetaData getMetaData() throws ResourceException {
      return new JNIResourceAdapterMetaData();
   }
   public void setReference(Reference r) {
      ref = r;
   }
```

```
   public Reference getReference() {
      return ref;
   };
}
```

JNI connection manager

Description: Responsible for pooling connections
File name: JNIConnectionManager.java

```
package com.ibm.redbook.scenario3.RA;

import java.io.Serializable;
import javax.resource.ResourceException;
import javax.resource.cci.Connection;
import javax.resource.spi.ConnectionManager;
import javax.resource.spi.ConnectionRequestInfo;
import javax.resource.spi.ManagedConnection;
import javax.resource.spi.ManagedConnectionFactory;
import javax.security.auth.Subject;

public class JNIConnectionManager implements ConnectionManager,Serializable {
   private Subject subject;
   public Object allocateConnection(
      ManagedConnectionFactory mcf,
      ConnectionRequestInfo cx) throws ResourceException {
      ManagedConnection mc = mcf.createManagedConnection(null, null);
      return mc.getConnection(null,null);
   }
}
```

JNI managed connection

Description: Implements the ManagedConnection class for the physical implemention of the connection
File name: JNIManagedConnection.java

```
package com.ibm.redbook.scenario3.RA;

import java.io.PrintWriter;
import java.io.Serializable;
import java.util.Vector;

import javax.resource.NotSupportedException;
import javax.resource.ResourceException;
import javax.resource.cci.Connection;
import javax.resource.spi.ConnectionEventListener;
```

```
import javax.resource.spi.ConnectionRequestInfo;
import javax.resource.spi.LocalTransaction;
import javax.resource.spi.ManagedConnection;
import javax.resource.spi.ManagedConnectionMetaData;
import javax.security.auth.Subject;
import javax.transaction.xa.XAException;
import javax.transaction.xa.XAResource;
import javax.transaction.xa.Xid;

public class JNIManagedConnection
   implements ManagedConnection,Serializable {
   private PrintWriter logWriter;
   private Vector conListeners = new Vector();
   private Object con;
   public JNIManagedConnection(
      Subject subject,
      ConnectionRequestInfo cxRequestInfo) {
   }
   public Object getConnection(
      Subject subject,
      ConnectionRequestInfo cxRequestInfo)
      throws ResourceException {
      return new JNIConnection(this);
   }
   public void destroy() throws ResourceException {
   }
   public void cleanup() throws ResourceException {
   }
   public void addConnectionEventListener(ConnectionEventListener listener) {
      conListeners.add(listener);
   }
   public void removeConnectionEventListener(ConnectionEventListener listener)
{
      conListeners.remove(listener);
   }
   public ManagedConnectionMetaData getMetaData() throws ResourceException {
      return new JNIManagedConnectionMetaData();
   }
   public LocalTransaction getLocalTransaction() throws NotSupportedException {
      throw new NotSupportedException("JNIManagedConnection");
   }
   public void setLogWriter(PrintWriter p) {
      logWriter = p;
   }
   public PrintWriter getLogWriter() {
      return logWriter;
   }
   public void associateConnection(Object o) {
      con = o;
```

```
    }
    public XAResource getXAResource() throws NotSupportedException {
        throw new NotSupportedException("JNIManagedConnection");
    }
}
```

JNI connection meta data

Description: Returns information (meta data) about the JNIConnection class
File name: JNIConnectionMetaData.java

```
package com.ibm.redbook.scenario3.RA;

import java.io.Serializable;
import javax.resource.ResourceException;
import javax.resource.cci.ConnectionMetaData;

public class JNIConnectionMetaData implements ConnectionMetaData,Serializable {
    public String getEISProductName() throws ResourceException {
        return (JNIManagedConnectionMetaData.PRODUCT_NAME);
    }
    public String getEISProductVersion() throws ResourceException {
        return (JNIManagedConnectionMetaData.PRODUCT_VERSION);
    }
    public String getUserName() throws ResourceException {
        throw new ResourceException("cannot get user name");
    }
}
```

JNI managed connection factory interface

Description: Implements the ManagedConnectionFactory interface
File name: JNIManagedConnectionFactory.java

```
package com.ibm.redbook.scenario3.RA;

import javax.resource.cci.ConnectionSpec;
import javax.resource.spi.ConnectionManager;
import javax.resource.spi.ConnectionRequestInfo;
import javax.resource.spi.ManagedConnection;
import javax.resource.spi.ManagedConnectionFactory;

import java.io.PrintWriter;
import java.io.Serializable;
import java.util.Iterator;

import javax.resource.ResourceException;
```

```
public class JNIManagedConnectionFactory implements
ManagedConnectionFactory,Serializable {
   private PrintWriter logWriter;
   private ConnectionManager conManager = new JNIConnectionManager();
   public JNIManagedConnectionFactory() {
   }
   public Object createConnectionFactory(ConnectionManager connectionManager)
      throws ResourceException {
      conManager = connectionManager;
      return new JNIConnectionFactory(this, conManager);
   }
   public void setUserName(String u) {
   }
   public void setPassword(String p) {
   }
   public String getUserName() {
      return null;
   }
   public String getPassword() {
      return null;
   }
   public Object createConnectionFactory() throws ResourceException {
      return new JNIConnectionFactory(this, conManager);
   }
   public ManagedConnection createManagedConnection(
      javax.security.auth.Subject subject,
      ConnectionRequestInfo cxRequestInfo)
      throws ResourceException {
      return new JNIManagedConnection(subject, cxRequestInfo);
   }
   public ManagedConnection matchManagedConnections(
      java.util.Set connectionSet,
      javax.security.auth.Subject subject,
      ConnectionRequestInfo cxRequestInfo)
      throws ResourceException {
      return null; //creates a new ManagedConnection
   }
   public void setLogWriter(PrintWriter p) {
      logWriter = p;
   }
   public PrintWriter getLogWriter() {
      return logWriter;
   }
}
```

JNI managed connection meta data

Description: JNIManagedConnection meta data
File name: JNIManagedConnectionMetaData.java

```
package com.ibm.redbook.scenario3.RA;

import java.io.Serializable;
import javax.resource.ResourceException;
import javax.resource.spi.ManagedConnectionMetaData;

public class JNIManagedConnectionMetaData
   implements ManagedConnectionMetaData,Serializable {
   public static final String PRODUCT_VERSION = "1.0";
   public static final String PRODUCT_NAME = "Department Server";
   public int getMaxConnections() throws ResourceException {
      throw new ResourceException("cannot get max connections");
   }
   public String getEISProductVersion() {
      return PRODUCT_VERSION;
   }
   public String getEISProductName() {
      return PRODUCT_NAME;
   }
   public String getUserName() throws ResourceException {
      throw new ResourceException("cannot get a user name");
   }
}
```

JNI resource adapter meta data

Description: JNI resource adapter meta data
File name: JNIResourceAdapterMetaData.java

```
package com.ibm.redbook.scenario3.RA;

import java.io.Serializable;
import javax.resource.cci.ResourceAdapterMetaData;

public class JNIResourceAdapterMetaData implements
ResourceAdapterMetaData,Serializable {

   public String getAdapterVersion() {
      return "1.0";
   }
   public String getAdapterVendorName() {
      return "IBM";
   }
```

```
   public String getAdapterName() {
      return "JNI Adapter without transaction and security support";
   }
   public String getAdapterShortDescription() {
      return "JNI Adapter";
   }
   public String getSpecVersion() {
      return "1.0";
   }
   public String[] getInteractionSpecsSupported() {
      return null;
   }
   public boolean supportsExecuteWithInputAndOutputRecord() {
      return false;
   }
   public boolean supportsExecuteWithInputRecordOnly() {
      return false;
   }
   public boolean supportsLocalTransactionDemarcation() {
      return false;
   }
}
```

Enterprise bean wrapper

Description: Enterprise bean wrapper for C application server
File name: DirectorySessionBeanImpl.java

```
package com.ibm.redbook.scenario3.EJB;

import javax.naming.InitialContext;
import javax.resource.cci.Connection;
import javax.resource.cci.ConnectionFactory;

import com.ibm.redbook.scenario3.RA.*;
public class DirectorySessionBeanImpl implements javax.ejb.SessionBean {
   private javax.ejb.SessionContext mySessionCtx;
   private Connection dept = null;
   public javax.ejb.SessionContext getSessionContext() {
      return mySessionCtx;
   }
   public void setSessionContext(javax.ejb.SessionContext ctx) {
      mySessionCtx = ctx;
   }
   public void ejbCreate() throws javax.ejb.CreateException {
   }
   public void ejbActivate() {
   }
```

```
   public void ejbPassivate() {
   }
   public void ejbRemove() {
   }
   public Connection getcon() throws Exception {
      try {
         InitialContext ic = new InitialContext();
         ConnectionFactory cf =(ConnectionFactory)
ic.lookup("java:comp/env/redbook_jni_connection");
         return cf.getConnection();
      } catch (Exception e) {
         System.out.println("Cannot get a connection "+e.getMessage());
         e.printStackTrace();
         throw e;
      }
   }
   public String getDepartment(String emp_name) {
      try {
         if (dept == null)
            dept = getcon();
         return ((JNIConnection) dept).getDepartment(emp_name);
      } catch (Exception e) {
         System.out.println("Couldn't create JNIConnection");
      }
      return null;
   }
   public int getGrade(String emp_name) {
      try {
         if (dept == null)
            dept = getcon();
         return ((JNIConnection) dept).getGrade(emp_name);
      } catch (Exception e) {
         System.out.println("Couldn't create JNIConnection");
      }
      return -1;
   }
   public double getSalary(String emp_name) {
      try {
         if (dept == null)
            dept = getcon();
         return ((JNIConnection) dept).getSalary(emp_name);
      } catch (Exception e) {
         System.out.println("Couldn't create JNIConnection");
      }
      return -1.0;
   }
}
```

Enterprise bean remote interface

Description: Remote interface for Enterprise Bean wrapper
File name: DirectorySessionBean.java

```
package com.ibm.redbook.scenario3.EJB;

import java.rmi.RemoteException;

import javax.ejb.EJBObject;

/**
 * Remote interface for Enterprise Bean: DirectorySessionBean
 */
public interface DirectorySessionBean extends EJBObject {
   String getDepartment(String emp_name) throws RemoteException;
   int getGrade(String emp_name)throws RemoteException;
   double getSalary(String emp_name) throws RemoteException;
}
```

Enterprise bean home interface

Description: Home interface for Enterprise Bean wrapper
File name: DirectorySessionHome.java

```
package com.ibm.redbook.scenario3.EJB;
/**
 * Home interface for Enterprise Bean: DirectorySessionBean
 */
public interface DirectorySessionHome extends javax.ejb.EJBHome {
   /**
    * Creates a default instance of Session Bean: DirectorySessionBean
    */
   public com.ibm.redbook.scenario3.EJB.DirectorySessionBean create()
      throws javax.ejb.CreateException, java.rmi.RemoteException;
}
```

Deployment descriptor for the application

Description: Deployment descriptor for the enterprise application
File name: application.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">
<application id="Application_ID">
   <display-name>Redbook Sample</display-name>
```

```
   <module id="EjbModule_1053352417969">
      <ejb>Redbook_Sample_EJB.jar</ejb>
   </module>
   <module id="ConnectorModule_1053549528151">
      <connector>Redbook_Sample_Connector.rar</connector>
   </module>
</application>
```

Deployment descriptor for the enterprise bean

Description: Deployment descriptor for the Enterprise Bean
File name: ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar id="ejb-jar_ID">
   <display-name>Redbook Sample EJB</display-name>
   <enterprise-beans>
      <session id="Directory">
         <ejb-name>Directory</ejb-name>
         <home>com.ibm.redbook.scenario3.EJB.DirectorySessionHome</home>
         <remote>com.ibm.redbook.scenario3.EJB.DirectorySessionBean</remote>

<ejb-class>com.ibm.redbook.scenario3.EJB.DirectorySessionBeanImpl</ejb-class>
         <session-type>Stateless</session-type>
         <transaction-type>Container</transaction-type>
         <resource-ref id="ResourceRef_1054820836658">
            <description></description>
            <res-ref-name>redbook_jni_connection</res-ref-name>
            <res-type>javax.resource.cci.ConnectionFactory</res-type>
            <res-auth>Application</res-auth>
            <res-sharing-scope>Unshareable</res-sharing-scope>
         </resource-ref>
      </session>
   </enterprise-beans>
   <assembly-descriptor>
      <security-role>
         <description>Directory Manager</description>
         <role-name>manager</role-name>
      </security-role>
      <security-role>
         <description>Directory Employee</description>
         <role-name>employee</role-name>
      </security-role>
      <method-permission>
         <role-name>manager</role-name>
         <method>
```

```
            <ejb-name>Directory</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getGrade</method-name>
            <method-params>
               <method-param>java.lang.String</method-param>
            </method-params>
         </method>
         <method>
            <ejb-name>Directory</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getSalary</method-name>
            <method-params>
               <method-param>java.lang.String</method-param>
            </method-params>
         </method>
      </method-permission>
      <method-permission>
         <role-name>employee</role-name>
         <method>
            <ejb-name>Directory</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getGrade</method-name>
            <method-params>
               <method-param>java.lang.String</method-param>
            </method-params>
         </method>
      </method-permission>
      <method-permission>
         <unchecked />
         <method>
            <ejb-name>Directory</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>getDepartment</method-name>
            <method-params>
               <method-param>java.lang.String</method-param>
            </method-params>
         </method>
      </method-permission>
   </assembly-descriptor>
</ejb-jar>
```

Deployment descriptor for the resource adapter

Description: Deployment descriptor for the JCA resource adapter
File name: ra.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE connector PUBLIC "-//Sun Microsystems, Inc.//DTD Connector 1.0//EN"
"http://java.sun.com/dtd/connector_1_0.dtd">
<connector>
   <display-name>Redbook Sample Connector</display-name>
   <description></description>
    <vendor-name>IBM</vendor-name>
     <spec-version>1.0</spec-version>
     <eis-type></eis-type>
     <version>1.0</version>
     <license>
        <description></description>
       <license-required>false</license-required>
    </license>
     <resourceadapter>

<managedconnectionfactory-class>com.ibm.redbook.scenario3.RA.JNIManagedConnecti
onFactory</managedconnectionfactory-class>

<connectionfactory-interface>javax.resource.cci.ConnectionFactory</connectionfa
ctory-interface>

<connectionfactory-impl-class>com.ibm.redbook.scenario3.RA.JNIConnectionFactory
</connectionfactory-impl-class>

<connection-interface>javax.resource.cci.Connection</connection-interface>

<connection-impl-class>com.ibm.redbook.scenario3.RA.JNIConnection</connection-i
mpl-class>
       <transaction-support>NoTransaction</transaction-support>
       <reauthentication-support>false</reauthentication-support>
     </resourceadapter>
</connector>
```

“C” application server

Description: C application server
File name: directory_impl.c

```
#include <stdio.h>
#include <string.h>
#include "directory_impl.h"
typedef struct {
```

```
      char *emp_name;
      char *dept_name;
   long grade_value;
   double salary_value;
} emp_entry_t;

emp_entry_t emp_table [] = {
     {"peter_morgan", "Accounting",1,24000.0},
      {"ruth_jones", "Marketing",2,38000.0},
      {"howard_stein","Finance",2,42000.0},
      {"fran_cooper", "Administration",1,27000.0},
      {"john_doe", "Training",1,18000.0},
   {NULL,NULL,-1,0}
};
DIRECTORY_EXPORT char *getDepartmentImpl(char *name) {
   emp_entry_t *emp_table_p=emp_table;
   int i=0;
      for (i=0;; i++) {
      if(emp_table_p->emp_name==NULL) return "unknown";
          if (! strcmp (name, emp_table_p->emp_name))
         return emp_table_p->dept_name;
      emp_table_p++;
   }
   return "unknown";
}
DIRECTORY_EXPORT long getGradeImpl(char *name) {
   emp_entry_t *emp_table_p=emp_table;
   int i=0;
      for (i=0;; i++) {
      if(emp_table_p->emp_name==NULL) return -1;
          if (! strcmp (name, emp_table_p->emp_name))
         return emp_table_p->grade_value;
      emp_table_p++;
   }
   return -1;
}
DIRECTORY_EXPORT double getSalaryImpl(char *name) {
   emp_entry_t *emp_table_p=emp_table;
   int i=0;
      for (i=0;; i++) {
      if(emp_table_p->emp_name==NULL) return -1.0;
          if (! strcmp (name, emp_table_p->emp_name))
         return emp_table_p->salary_value;
      emp_table_p++;
   }
   return -1.0;
}
```

JNI wrapper for the application server

Description: JNI wrapper for the C application server
File name: directory_jni_impl.c

```
#include <stdio.h>
#include <string.h>
#include <jni.h>
#include "com_ibm_redbook_scenario3_RA_JNIConnection.h"
#include "directory_impl.h"
JNIEXPORT jstring JNICALL
Java_com_ibm_redbook_scenario3_RA_JNIConnection_getDepartmentJNI(
   JNIEnv *env,
   jobject obj,
   jstring s
)
{
   const char *name;
   char *n;
   printf("In JNI Impl\n");
   name=(*env)->GetStringUTFChars(env,s,0);
   printf("In JNI Impl\n");
   n=getDepartmentImpl((char *)name);
   (*env)->ReleaseStringUTFChars(env,s,name);
   return (*env)->NewStringUTF(env,n);
}
JNIEXPORT jint
   JNICALL Java_com_ibm_redbook_scenario3_RA_JNIConnection_getGradeJNI(
      JNIEnv *env,
      jobject obj,
      jstring s
)
{
   const char *name=(*env)->GetStringUTFChars(env,s,0);
   const int g=getGradeImpl((char *)name);
   (*env)->ReleaseStringUTFChars(env,s,name);
   return (jint) g;
}
JNIEXPORT jdouble
   JNICALL Java_com_ibm_redbook_scenario3_RA_JNIConnection_getSalaryJNI(
      JNIEnv *env,
      jobject obj,
      jstring s
)
{
   const char *name=(*env)->GetStringUTFChars(env,s,0);
   const double sal=getSalaryImpl((char *)name);
```

```
   (*env)->ReleaseStringUTFChars(env,s,name);
   return (jdouble) sal;
}
```

Header file for application server

Description: Header file for application server
File name: directory_impl.h

```
#ifndef DIRECTORY_EXPORT
#define DIRECTORY_EXPORT
#endif
DIRECTORY_EXPORT char *getDepartmentImpl(char *name);
DIRECTORY_EXPORT double getSalaryImpl(char *name);
DIRECTORY_EXPORT long getGradeImpl(char *name);
```

CORBA C++ client

Description: The CORBA C++ client
File name: corba_client.cpp

```
#include <stdlib.h>
#include <stdio.h>
#include "DirectorySessionHome.hh"
#include <CosNaming.hh>
#include <corba.h>
#include <vtlib.h>
com::ibm::redbook::scenario3::EJB::DirectorySessionBean_ptr bptr;
void show_usage(int argc, char *argv[]) {
   printf("Usage is: %s <-function {0|1|2}> <-employee emp_name>",argv[0]);
   exit(1);
}
void execute_query(int argc, char **argv) {
   int i=0;
   int query_type=0;
   enum query_type {DEPT,GRADE,SALARY};
   CORBA::WStringValue  *corba_name=com::ibm::ws::VtlUtil::toWStringValue("");
   char *emp_name="";
   CORBA::Long grade;
   CORBA::Double salary;
   CORBA::WStringValue *dept_name;
   for(i=0;i<argc;i++) {
      if(strcmp(argv[i],"-function")==0) {
         if(i<argc-1) {
            query_type=atoi(argv[i+1]);
            i++;
         }
```

```
      }
      if(strcmp(argv[i],"-employee")==0) {
         if(i<argc-1) {
            emp_name=argv[i+1];
            corba_name=com::ibm::ws::VtlUtil::toWStringValue(argv[i+1]);
            i++;
         }
      }
      if(strcmp(argv[i],"-?")==0) show_usage(argc,argv);
   }
   switch(query_type) {
      case DEPT: {
         dept_name=bptr->getDepartment(corba_name);
         printf (
            "Department for %s is: %s\n",
            strcmp(emp_name,"")!=0?emp_name:"current user",
            com::ibm::ws::VtlUtil::WStringValueToString(dept_name)
         );
         break;
      }
      case GRADE: {
         grade=bptr->getGrade(corba_name);
         if(grade<0) {
            printf (
               "Grade for %s is: unknown\n",
               strcmp(emp_name,"")!=0?emp_name:"current user",
               grade
            );
         }
         else {
            printf (
               "Grade for %s is: %i\n",
               strcmp(emp_name,"")!=0?emp_name:"current user",
               grade
            );
         }
         break;
      }
      case SALARY:{
         salary=bptr->getSalary(corba_name);
         if(salary<0) {
            printf (
               "Salary for %s is: unknown\n",
               strcmp(emp_name,"")!=0?emp_name:"current user",
               salary
            );
         }
         else {
            printf (
```

```
            "Salary for %s is: %8.2f\n",
            strcmp(emp_name,"")!=0?emp_name:"current user",
            salary
         );
      }
      break;
   }
   default: {
      printf("Invalid query type %d\n",query_type);
      show_usage(argc,argv);
      break;
   }
  }
}
int main(int argc, char *argv[]) {
   CORBA::Object_ptr objPtr;
   com::ibm::redbook::scenario3::EJB::DirectorySessionHome_ptr liptr;
   CORBA::ORB_ptr orbPtr = CORBA::ORB_init ( argc, argv, "DSOM" );
   if ( CORBA::is_nil(orbPtr) )
   {
      cerr << "Error initializing the ORB!" << endl;
      return -1;
   }
   try
   {
      Properties *props = CORBA::ORB::get_properties();
      const char *boothost = props->getProperty
("com.ibm.CORBA.bootstrapHostName");
      const char *bootport=props->getProperty( "com.ibm.CORBA.bootstrapPort");
      const char
*servername=props->getProperty("com.ibm.websphere.serverName");
      const char *ejbname=props->getProperty("com.ibm.websphere.EJBName");
      char *home_url=(char *)malloc(512);
      strcpy(home_url,"corbaname::");
      if(boothost==NULL||bootport==NULL||servername==NULL||ejbname==NULL) {
         printf("Missing property in WASPROPS file\n");
         exit(1);
      }
      strcat(home_url,boothost);
      strcat(home_url,":");
      strcat(home_url,bootport);
      strcat(home_url,"/NameService#nodes/");
      strcat(home_url,boothost);
      strcat(home_url,"/servers/");
      strcat(home_url,servername);
      strcat(home_url,"/");
      strcat(home_url,ejbname);
      objPtr = orbPtr->string_to_object(home_url);
      liptr=com::ibm::redbook::scenario3::EJB::DirectorySessionHome\
```

```
            ::_narrow( objPtr);
         bptr=liptr->create();
      }
      catch (CORBA::UserException &ue)
      {
         cerr << "Caught a User Exception: " << ue.id() << endl;
         return -1;
      }
      catch (CORBA::SystemException &se)
      {
         cerr<<"Caught a System Exception: "<<se.id()<< ": "<<se.minor()<<endl;
         return -1;
      }
      execute_query(argc,argv);
      CORBA::release (bptr);
      CORBA::release ( orbPtr );
      return(0);
   }
```

Properties file for CORBA C++ client

Description: Properties file for the CORBA C++ client (AIX)
File name: WSEJBClient.props

```
com.ibm.CORBA.bootstrapPort=2809
com.ibm.CORBA.translationEnabled=1
com.ibm.CORBA.nativeWCharCodeset=UCS2
com.ibm.websphere.serverName=server1
com.ibm.websphere.EJBName=ejb/com/ibm/redbook/scenario3/EJB/DirectoryHome
com.ibm.CORBA.securityEnabled=yes
com.ibm.ssl.keyFile=/usr/WebSphere/AppServer/etc/ITSOKeyRingFile.KDB
com.ibm.ssl.keyPassword=WebAS
com.ibm.CSI.performTLClientAuthenticationSupported=yes
com.ibm.CSI.performTransportAssocSSLTLSSupported=yes
com.ibm.CORBA.initServices=security
com.ibm.CORBA.dllName=libwassccl.so
com.ibm.CORBA.securityTraceLevel=1
com.ibm.CSI.performMessageIntegritySupported=yes
```

D

Scenario 4: Source code listings

This appendix lists the complete source code that was explained in pieces in Chapter 11, “Scenario 4: Secure RPC application #2” on page 247. Specifically, this appendix contains:

- The source code of the client and server programs with DCE dependencies
- The source code of the revised client and server programs without DCE dependencies
- The makefiles used for these programs
- Additional source code modules for authorization
- Header files
- Application configuration files

Refer to Appendix E, “Additional material” on page 417 for instructions for downloading the source code samples included in this appendix.

© Copyright IBM Corp. 2003. All rights reserved.

Application with DCE dependencies

The application with DCE dependencies in this scenario is the same as in the previous scenario; refer to "Application with DCE dependencies" on page 364. The application logic of the application in this scenario with DCE dependencies is the same as in scenario 2. (See "Application server logic" on page 341.)

Revised application without DCE dependencies

The files are:

- build.bat contains information about how the Java code is compiled and the jar files are built.
- MirrorClient.java contains the Java code of the application client. Connects to an EJB and takes user input in a loop until 'exit' is entered.
- MirrorBean.java contains the Java code of the stateless session bean used in this scenario.
- MirrorHome.java contains the Java defintion of MirrorBeans home interface.
- Mirror.java contains the Java defintion of MirrorBeans remote interface.
- MirrorException.java contains the Java code for the excpetion, that can be thrown by the remote interface method of MirrorBean.
- ejb-jar.xml is the deployment descriptor for MirrorBean, like it is created by the WebSphere Application Server's application assembly tool.
- application-client.xml is the deployment descriptor for application client, like it is created by the WebSphere Application Server's application assembly tool.
- application.xml is the deployment descriptor for mirror application, like it is created by the WebSphere Application Server's application assembly tool.
- sas.client.props contains SAS (Secure Associatation Service) definitions to be used by the J2EE client. Directory path for key files makes the current file dependend on the Windows platform.

Build script to create class and jar files

Description: Build script for the mirror application jar files
File name: build.bat

```
set classpath=%classpath%;%JAVA_HOME%\lib\j2ee.jar
set classpath=%classpath%;\TI\SE-S001 DCE  Replacement\Coding\scenario4\nondce
set path=%path%;%JAVA_HOME%\bin
```

```
javac .\mirrorClient\*.java
javac .\mirrorServer\*.java
del MirrorServer.jar
del MirrorClient.jar
jar -cvf MirrorServer.jar .\mirrorServer\*.class
jar -cvf MirrorClient.jar .\mirrorClient\*.class
```

Java client program

Description: Java application client that uses MirrorBean EJB
File name: MirrorClient.java

```
package mirrorClient;

import javax.ejb.*;
import javax.naming.*;
import java.rmi.*;
import javax.rmi.PortableRemoteObject;
import java.util.*;
import java.io.*;

import mirrorServer.*;

public class MirrorClient {

   public static void main(String[] args) throws Exception {

      MirrorHome home   = null;
      String message    = new String();
      String cmdArgs[]  = null;
      Mirror mirror     = null;

      /* Create input stream for reading terminal input from the user */

      InputStreamReader converter = new InputStreamReader(System.in);
      BufferedReader in = new BufferedReader(converter);

      try {

         Context initial = new InitialContext();
         Object obj = initial.lookup("java:comp/env/MirrorHome");

         home = (MirrorHome)PortableRemoteObject.narrow(obj, MirrorHome.class);

         mirror = home.create();

      } catch (java.rmi.AccessException ex) {
         System.err.println("Access Denied");
```

```
        return;
      } catch (NamingException ex) {
        System.err.println("Could not find Server");
        return;
      } catch (Exception ex) {
        System.err.println("Unexpected exception binding to server:");
        ex.printStackTrace();
        return;
      }

      while (true) {

        try {
          System.out.print("Ready> ");
          message = in.readLine();
          if (message.compareTo("exit") == 0) break;

          message = mirror.reflect(message);

          System.out.println("reflected string is: " + message);

        } catch (java.rmi.AccessException ex) {
          System.err.println("Access Denied");
        } catch (MirrorException ex) {
          System.err.println(ex.getMessage());
        } catch (Exception ex) {
          System.err.println("Unexpected exception processing request:");
          ex.printStackTrace();
        }
      }
    }
  }
```

Java stateless session bean

Description: Java code for MirrorBean EJB
File name: MirrorBean.java

```
package mirrorServer;

import java.util.*;
import javax.ejb.*;
import javax.naming.*;

public class MirrorBean implements javax.ejb.SessionBean {

   private javax.ejb.SessionContext sessionContext;
```

```
public MirrorBean() {}

public void ejbCreate() {
   System.out.println("ejbCreate()");
}

public void ejbRemove() {
   System.out.println("ejbRemove()");
}

public void ejbActivate() {
   System.out.println("ejbActivate()");
}

public void ejbPassivate() {
   System.out.println("ejbPassivate()");
}

public void setSessionContext(javax.ejb.SessionContext sessionContext) {
   System.out.println("setSessionContext()");
   this.sessionContext = sessionContext;
}

public String reflect(String message) throws MirrorException {

   System.out.println("received message: " + message);

   //
   // example how authz can be performed in code
   //
   java.security.Principal principal = sessionContext.getCallerPrincipal();
   java.lang.String callerId = principal.getName();
   System.out.println("received call from: " + callerId);

   boolean isMgr = sessionContext.isCallerInRole("manager");
   boolean isUser = sessionContext.isCallerInRole("user");
   if(!isMgr) {
      if (isUser) {
         System.out.println("Sorry, " + callerId + " you are just a user");
         throw new MirrorException
            ("Sorry, " + callerId + " you are just a user");
      }
      //
      // Allow access for other roles, to keep security configurable
      //
   }
   //
   // perform the actual 'business logic'
   //
```

```
      StringBuffer tmp = new StringBuffer(message);

      return (tmp.reverse()).toString();
   }
}
```

Java EJB home interface

Description: Home Interface of MirrorBean EJB
File name: MirrorHome.java

```
package mirrorServer;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface MirrorHome extends EJBHome
{
   Mirror create() throws RemoteException, CreateException;
}
```

Java EJB remote interface

Description: Remote Interface of MirrorBean EJB
File name: Mirror.java

```
package mirrorServer;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Mirror extends EJBObject
{
   public String reflect(String message)
               throws MirrorException, RemoteException;
}
```

Java application exception

Description: Exception, used by MirrorBean EJB
File name: MirrorException.java

```
package mirrorServer;

public class MirrorException extends Exception {

   public MirrorException() {}
   public MirrorException(String msg) {
      super(msg);
   }
}
```

EJB deployment descriptor

Description: Generated deployment descriptor for MirrorBean EJB
File name: ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN" "http://java.sun.com/dtd/ejb-jar_2_0.dtd">

   <ejb-jar id="ejb-jar_ID">
      <display-name>MirrorBean</display-name>
      <enterprise-beans>
         <session id="Session_1054051282734">
            <display-name>MirrorBean</display-name>
            <ejb-name>MirrorBean</ejb-name>
            <home>mirrorServer.MirrorHome</home>
            <remote>mirrorServer.Mirror</remote>
            <ejb-class>mirrorServer.MirrorBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
            <security-role-ref id="SecurityRoleRef_1054051785469">
               <role-name>manager</role-name>
               <role-link>manager</role-link>
            </security-role-ref>
            <security-role-ref id="SecurityRoleRef_1054215739406">
               <role-name>user</role-name>
               <role-link>user</role-link>
            </security-role-ref>
         </session>
      </enterprise-beans>
      <assembly-descriptor id="AssemblyDescriptor_1054051282797">
         <security-role id="SecurityRole_1054051785469">
            <role-name>manager</role-name>
```

```
            </security-role>
            <security-role id="SecurityRole_1054215739406">
               <role-name>user</role-name>
            </security-role>
            <method-permission id="MethodPermission_1054054624016">
               <description>reflectAccess:+:</description>
               <role-name>manager</role-name>
               <role-name>user</role-name>
               <method id="MethodElement_1054215739406">
                  <ejb-name>MirrorBean</ejb-name>
                  <method-name>reflect</method-name>
                  <method-params>
                     <method-param>java.lang.String</method-param>
                  </method-params>
               </method>
            </method-permission>
            <method-permission id="MethodPermission_1054215739406">
               <description>createAccess:+:</description>
               <role-name>user</role-name>
               <role-name>manager</role-name>
               <method id="MethodElement_1054215739407">
                  <ejb-name>MirrorBean</ejb-name>
                  <method-name>create</method-name>
                  <method-params>
                  </method-params>
               </method>
            </method-permission>
         </assembly-descriptor>
         <ejb-client-jar>C:\MirrorClient.jar</ejb-client-jar>
      </ejb-jar>
```

Application client deployment descriptor

Description: Generated deployment descriptor for application client
File name: application-client.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application-client PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE
Application Client 1.3//EN"
"http://java.sun.com/dtd/application-client_1_3.dtd">

   <application-client id="Application-client_ID">
      <display-name>MirrorClient</display-name>
      <ejb-ref id="EjbRef_1054051283203">
         <ejb-ref-name>MirrorHome</ejb-ref-name>
         <ejb-ref-type>Session</ejb-ref-type>
         <home>mirrorServer.MirrorHome</home>
         <remote>mirrorServer.Mirror</remote>
```

```
      </ejb-ref>
   </application-client>
```

Application deployment descriptor

Description: Generated deployment descriptor
File name: application.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application
1.3//EN" "http://java.sun.com/dtd/application_1_3.dtd">

   <application id="Application_ID">
      <display-name>MirrorApp</display-name>
      <module id="EjbModule_1054051281578">
         <ejb>MirrorBean.jar</ejb>
      </module>
      <module id="JavaClientModule_1054051281688">
         <java>MirrorClt.jar</java>
      </module>
      <security-role id="SecurityRole_1054051281719">
         <role-name>manager</role-name>
      </security-role>
      <security-role id="SecurityRole_1054054623938">
         <role-name>user</role-name>
      </security-role>
   </application>
```

Application client security properties

Description: SAS settings to be used by Windows Java clients
File name: sas.client.props

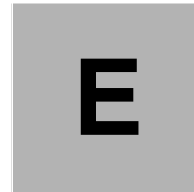
```
################################################################################
#
#                              SAS Properties File
#
#  This file contains properties that are used by the Secure Association
#  Services (SAS) component of the WebSphere Application Server product.
#  SAS executes on WebSphere java servers and client systems with java
#  applications that access WebSphere servers.
#
#  ** SAS/CSIv2 Trace Instructions **
#
#  Note:  To enable logging of trace on the application client, add the
#  following property to the startup script: -DtraceSettingsFile=filename.
#  Do not specify filename as a fully qualified path and filename, just
```

```
#  specify the filename.  The file must exist in the classpath to be loaded.
#  A sample file is provided in <was_root>/properties/TraceSettings.properties.
#
#  There are two related functions provided by this file:
#
#    1.traceFileName property
#      This should be set to the fully qualified name of a file to which you
#      want
#      output written. For example, traceFileName=c:\\MyTraceFile.log. This
#      property must be specified, otherwise no visible output is generated.
#    2.Trace string
#      To enable SAS/CSIv2 trace, specify the trace string: SASRas=all=enabled
#
#  If you only want to trace specific classes, you can specify a trace filter
#  by adding the property com.ibm.CORBA.securityTraceFilter=<comma-separated
#  class names>
#
#  Example:  com.ibm.CORBA.securityTraceFilter=SecurityConnectionInterceptor,
#  CSIClientRI, SessionManager
#
#  ** Encoding Passwords in this File **
#
#  The PropFilePasswordEncoder utility may be used to encode passwords in a
#  properties file. To edit an encoded password, replace the whole password
#  string (including the encoding tag {...}) with the new password and then
#  encode the password with the PropFilePasswordEncoder utility. Refer to
#  product documentation for additional information.
#
###############################################################################

#------------------------------------------------------------------------------
# Client Security Enablement
#
# - security enabled status  ( false, true [default] )
#------------------------------------------------------------------------------
com.ibm.CORBA.securityEnabled=true

#------------------------------------------------------------------------------
# RMI/IIOP Authentication Protocol (sas, csiv2, both [default])
#
# Specify "both" when communicating with 5.0x and previous release servers.
# Specify "csiv2" when communicating with only 5.0x servers.
# Specify "sas" when communicating with only previous release servers.
#------------------------------------------------------------------------------
com.ibm.CSI.protocol=csiv2

#------------------------------------------------------------------------------
# Authentication Configuration
#
```

```
# - authenticationTarget      (BasicAuth [default], this is the only supported
#                              selection
#                             on a pure client for this release.  This is for
#                              message
#                              layer authentication only, SSL client
#                              certificate authentication
#                              is configured below under CSIv2 configuration.)
# - authenticationRetryEnabled (enables authentication retries if login fails
#                              when loginSource=prompt or stdin)
# - authenticationRetryCount   (the number of times to retry)
# - source                     (prompt [default], properties, keyfile, stdin,
#                              none)
# - timeout                    (prompt timeout, specified in seconds, 0 min to
#                              600 max [default 300])
# - validateBasicAuth          (determines if immediate authentication after
#                              uid/pw login,
#                              or wait for method request to send uid/pw
#                              to server,
#                              setting this to false gives the previous
#                              release behavior.)
# - securityServerHost         (when validateBasicAuth=true, this property
#                              might need to be set
#                              in order for security code to lookup
#                              SecurityServer.  Needs to be set to
#                              any running WebSphere server host in the cell
#                              you are authenticating to.
# - securityServerPort         (when validateBasicAuth=true, this property
#                              might need to be set
#                              in order for security code to lookup
#                              SecurityServer.  Needs to be set to
#                              the bootstrap port of the host chosen above.
# - loginUserid                (must be set if login source is "properties" )
# - loginPassword              (must be set if login source is "properties" )
# - principalName              (format: "realm/userid", only needed in cases
#                              where realm
#                              is required. Typically the realm is already
#                              provided by the
#                              server via the IOR and this property is not
#                              necessary).
#
#------------------------------------------------------------------------------
com.ibm.CORBA.authenticationTarget=BasicAuth
com.ibm.CORBA.authenticationRetryEnabled=true
com.ibm.CORBA.authenticationRetryCount=3
com.ibm.CORBA.validateBasicAuth=true
com.ibm.CORBA.securityServerHost=
com.ibm.CORBA.securityServerPort=
com.ibm.CORBA.loginTimeout=300
com.ibm.CORBA.loginSource=prompt
```

```
# RMI/IIOP user identity
com.ibm.CORBA.loginUserid=
com.ibm.CORBA.loginPassword=
com.ibm.CORBA.principalName=

#-------------------------------------------------------------------------------
# CSIv2 Configuration (see InfoCenter for more information on these
properties).
#
# This is where you enable SSL client certificate authentication.  Must also
# specify a valid SSL keyStore below with a personal certificate in it.
#-------------------------------------------------------------------------------

# Does this client support stateful sessions?
com.ibm.CSI.performStateful=true

# Does this client support/require BasicAuth (userid/password) client
# authentication?
com.ibm.CSI.performClientAuthenticationRequired=false
com.ibm.CSI.performClientAuthenticationSupported=false

# Does this client support/require SSL client authentication?
com.ibm.CSI.performTLClientAuthenticationRequired=false
com.ibm.CSI.performTLClientAuthenticationSupported=true

# Note: You can perform BasicAuth (uid/pw) and SSL client authentication
# (certificate)
# simultaneously, however, the BasicAuth identity will always take
# precedence at the server.

# Does this client support/require SSL connections?
com.ibm.CSI.performTransportAssocSSLTLSRequired=false
com.ibm.CSI.performTransportAssocSSLTLSSupported=true

# Does this client support/require 40-bit cipher suites when using SSL?
com.ibm.CSI.performMessageIntegrityRequired=false
com.ibm.CSI.performMessageIntegritySupported=true
# Note: This property is only valid when SSL connections are supported or
# required.

# Does this client support/require 128-bit cipher suites when using SSL?
com.ibm.CSI.performMessageConfidentialityRequired=false
com.ibm.CSI.performMessageConfidentialitySupported=true
# Note: This property is only valid when SSL connections are supported or
# required.

#-------------------------------------------------------------------------------
# SSL Configuration
```

```
#
# - protocol                          (SSL [default], SSLv2, SSLv3, TLS, TLSv1)
# - keyStoreType                      (JKS [default], JCEK, PKCS12)
# - trustStoreType                    (JKS [default], JCEK, PKCS12)
# - keyStore and trustStore           (fully qualified path to file)
# - keyStoreClientAlias
#                 (string specifying ssl certificate alias to use from keyStore)
# - keyStorePassword and trustStorePassword
#                  (string specifying password - encoded or not)
# - cipher suites
#                 (refer to InfoCenter for valid ciphers)
#
#    com.ibm.ssl.enabledCipherSuites=enabled_cipher_suites
#
#    Note: The com.ibm.ssl.enabledCipherSuites property defines the cipher
#          suites used for the SSL session. If this property is defined, it
#          overrides the default cipher suites defined for the specified QOP.
#
#------------------------------------------------------------------------------
com.ibm.ssl.protocol=SSL
com.ibm.ssl.keyStoreType=JKS
com.ibm.ssl.keyStore=C:/Program
Files/WebSphere/AppClient/etc/ITSOClientKeyFile.jks
com.ibm.ssl.keyStorePassword=WebAS
com.ibm.ssl.trustStoreType=JKS
com.ibm.ssl.trustStore=C:/Program
Files/WebSphere/AppClient/etc/ITSOClientTrustFile.jks
com.ibm.ssl.trustStorePassword=WebAS
com.ibm.ssl.keyStoreClientAlias=itso client
#------------------------------------------------------------------------------
# Quality of Protection for the IBM protocol
#
# - perform  ( high [default], medium, low )
#------------------------------------------------------------------------------
com.ibm.CORBA.standardPerformQOPModels=high

#------------------------------------------------------------------------------
# CORBA Request Timeout (used when getting NO_RESPONSE exceptions, typically
#                        during high-stress loads.  Specify on all processes
#                        involved in the communications.)
#
# - timeout         (specified in seconds [default 180], 0 implies no timeout)
#
#    com.ibm.CORBA.requestTimeout=180
#------------------------------------------------------------------------------
com.ibm.CORBA.requestTimeout=180
```



Additional material

This book contains program code examples for the DCE replacement example scenario 1 through 4. The source code for these example scenarios is listed in the previous appendixes, and it can be downloaded as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG246935`

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select **Additional materials** and open the directory that corresponds with the redbook form number SG246935.

© Copyright IBM Corp. 2003. All rights reserved.

Using the Web material

The additional Web material that accompanies this book includes the following four files:

File name	*Description*
6935_Scenario1.zip	Zipped code samples for scenario 1 of this book. (See Chapter 8, “Scenario 1: GSS-API application” on page 125 and Appendix A, “Scenario 1: Source code listings” on page 267.)
6935_Scenario2.zip	Zipped code samples for scenario 2 of this book. (See Chapter 9, “Scenario 2: Non-secure RPC application” on page 171 and Appendix B, “Scenario 2: Source code listings” on page 331.)
6935_Scenario3.zip	Zipped code samples for scenario 3 of this book. (See Chapter 10, “Scenario 3: Secure RPC application #1” on page 195 and Appendix C, “Scenario 3: Source code listings” on page 363.)
6935_Scenario4.zip	Zipped code samples for scenario 4 of this book. (See Chapter 11, “Scenario 4: Secure RPC application #2” on page 247 and Appendix D, “Scenario 4: Source code listings” on page 403.)

Once unpacked, each of these files contains two subdirectories: dce and nondce. The dce subdirectory contains the source code of the DCE dependent sample application, while the nondce directory contains the application without DCE dependencies. The filenames correspond to the filenames that are used in the previous appendixes.

Each subdirectory contains a Readme.txt file that you should read, as it contains additional information pertinent to the sample source code.

System requirements for downloading the Web material

The downloaded material consists of source code that requires only a few kilobytes of disk space. For building and running the sample applications, refer to the system requirements in the product documentation for the respective development and deployment environments and products.

How to use the Web material

Create a subdirectory (folder) on your workstation and unzip the contents of the Web material zip file into this folder. Then, follow the instructions in this book or in the Readme.txt files to build and run the sample applications.

Abbreviations and acronyms

AAT	Application Assembly Tool
ACL	Access Control List
API	Application Programming Interface
aznAPI	Authorization API
BOA	Basic Object Adapter
CDS	Cell Directory Service
CLI	Command Line Interface
CORBA	Common Object Request Broker Architecture
COSINE	Cooperation for Open Systems Interconnection Networking in Europe
CRL	Certificate Revocation List
CSIv2	Common Secure Interoperability Version 2
DAP	Directory Access Protocol
DCE	Distributed Computing Environment
DFS	Distributed File System
DIT	Directory Information Tree
DMT	Directory Management Tool
DN	Distinguished Name
DNS	Domain Name Service
DTS	Distributed Time Service
EIS	Enterprise Information Systems
EJB	Enterprise JavaBeans
ERA	Extended Registry Attributes
GDA	Global Directory Agent
GINA	Graphical Interface for Network Access
GIOP	General Inter-ORB Protocol
GSS-API	Generic Security Services Application Programming Interface
HPFS	High Performance File System
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines Corporation
IDE	Integrated Development Environment
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
IIOP	Internet Inter-ORB Protocol
INS	Interoperable Naming Service
IOR	Interoperable Object References
ITSO	International Technical Support Organization
J2EE	Java 2, Enterprise Edition
J2SE	Java 2, Standard Edition
JAAS	Java Authentication and Authorization Service
JAXP	Java API for XML Processing
JCA	Java Connector Architecture
JDBC	Java Database Connectivity
JDK	Java Development Kit
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JNI	Java Naming Interface
JVM	Java Virtual Machine
KDC	Key Distribution Center
LDAP	Lightweight Directory Access Protocol

© Copyright IBM Corp. 2003. All rights reserved.

LTPA	Lightweight Third-Party Authentication
NSA	National Security Agency
NTP	Network Time Protocol
OMG	Open Management Group
ORB	Object Request Broker
PAC	Privilege Attribute Certificate
PAM	Pluggable Authentication Module
PKI	Public Key Infrastructure
POP	Protected Object Policies
PSSP	Parallel System Support Program
QoS	Quality of Service
RDN	Relative Distinguished Name
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAS	Security Attribute Service
SNTP	Simple Network Time Protocol
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
SSO	Single Sign-On
SSPI	Security Support Provider Interface
SWAM	Simple WebSphere Authentication Mechanism
TGT	Ticket-Granting Ticket
TLS	Transport Layer Security
UDDI	Universal Description Discovery and Integration
UUID	Unique Universal Identifier
WAS	WebSphere Application Server
XML	eXtensible Markup Language

Related publications

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

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 423. Note that some of the documents referenced here may be available only in softcopy.

- *Enterprise Business Portals with IBM Tivoli Access Manager*, SG24-6556
- *Enterprise Business Portals II with IBM Tivoli Access Manager*, SG24-6885
- *IBM Tivoli Access Manager for e-business*, REDP3677
- *IBM WebSphere Application Server V5.0 System Management and Configuration*, SG24-6195
- *IBM WebSphere V5.0 Security*, SG24-6573
- *LDAP Implementation Cookbook*, SG24-5110
- *Understanding LDAP,* SG24-4986

Other publications

These publications are relevant as further information sources:

- John Shirley, et al., *Guide to Writing DCE Applications,* O'Reilly & Associates, Inc., 1994, ISBN 1565920457
- Michi Henning, et al., *Advanced CORBA Programming with C++,* Addison-Wesley Longmann, Inc., 1999, ISBN 0201379279
- Wei Hu, *DCE SECURITY Programming,* O'Reilly & Associates, Inc., 1995, ISBN 1565921348

© Copyright IBM Corp. 2003. All rights reserved.

Online resources

These Web sites and URLs are also relevant as further information sources:

- DCE Overview

  http://www.opengroup.org

- IBM DCE Version 3.2 for AIX and Solaris: Introduction to DCE

  http://www.ibm.com/software/network/dce/library/publications/dceintro/dceintro.pdf

- IBM DCE Version 3.2 for AIX and Solaris: DCE Security Registry and LDAP Integration Guide

  http://www.ibm.com/software/network/dce/library/publications/ldaprgy/ldaprgy.pdf

- The Common Security Request Broker: Architecture and Specification

  http://www.omg.org

- CORBA Naming Service Specification

  http://www.omg.org/docs/formal/02-09-02.pdf

- IBM WebSphere Application Server Enterprise, Version 5, Common Object Request Broker Architecture (CORBA)

  ftp://ftp.software.ibm.com/software/webserver/appserv/library/corba.pdf

- IBM WebSphere Application Server Enterprise, Verson 5, Applications

  ftp://ftp.software.ibm.com/software/webserver/appserv/library/wasv5ee_apps.pdf

- The Common Security Interoperability Version 2 Specification

  http://www.omg.org/docs/ptc/01-06-17.pdf

- C Language Mapping Specification

  http://www.omg.org/technology/documents/formal/c_language_mapping.htm

- Java Language Mapping to OMG IDL

  http://www.omg.org/technology/documents/formal/java_language_mapping_to_omg_idl.htm

- Java Internationalization Tutorial

  http://java.sun.com/docs/books/tutorial/i18n/index.html

- OMG IDL to Java Language Mapping

  http://www.omg.org/technology/documents/formal/omg_idl_to_java_language_mapping.htm

- IBM Network Authentication Service Version 1.3 for AIX, Administrator's and User's Guide

  Available on the AIX 5L Expansion Pack CD-ROM

- IBM Network Authentication Service Version 1.3 for AIX, Application Development Reference

  Available on the AIX 5L Expansion Pack CD-ROM

- AIX 5L Version 5.1 General Programming Concepts: Writing and Debugging Programs

  http://publibn.boulder.ibm.com/cgi-bin/ds_form?lang=en_US&viewset=AIX

- AIX 5L Version 5.1 Technical Reference: Base Operating System and Extensions, Volume 2

  http://publibn.boulder.ibm.com/cgi-bin/ds_form?lang=en_US&viewset=AIX

- IBM Tivoli Access Manager Base Administrator's Guide

  http://publib.boulder.ibm.com/tividd/td/ITAME/GC23-4684-00/en_US/PDF/adminmst.pdf

- Solaris Internationalization Guide For Developers

  http://docs.sun.com/db/doc/802-5878?q=catopen

- Sun Solaris Product Documentation, Basic Library Functions, syslog(3C)

  http://docs.sun.com/db/doc/806-0627/6j9vhfn8g?a=view

- Writing Multilingual User Interface Applications

  http://www.microsoft.com/globaldev/handson/dev/muiapp.mspx

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Index

Numerics
3DES 169

A
AAT 224, 255
Access Control List, see ACL
ACL 7, 53, 94, 134, 159
ACL manager 136
AcquireCredentialsHandle 153
additional material 417
AIX 26–27, 30, 48, 60, 126, 144, 151
AIX Parallel System Support Program (PSSP) 4
applet 35, 38, 74
Application Assembly Tool (AAT) 224, 255
applications in C/C++ 21, 52, 74
auditing 8, 31, 49, 52
authentication 7, 37, 53, 126
  basic 37, 40, 70
  certificate 40, 70, 73, 114, 117, 119, 195, 247
  credential token 40
  form-based 71
  strong 117
authorization 7, 37, 53, 126
  role-based 195, 247
azn_creds_create 156
azn_decision_access_allowed 157
azn_id_get_creds 156
aznAPI 22, 48, 53, 128, 152, 168

B
backing store 38, 56
backing store service 10
basic authentication 37, 40, 70
BasicAuth 223
bean, see enterprise bean
binary structure of ERA 105
binding 6, 65, 173, 178
  indirect 176
  usage 184
binding handle 178, 364
bootstrap host 45
bootstrap port 45
bootstrapHostName 183
bootstrapPort 183

C
C/C++ applications 21, 52, 74
CDS 9, 136, 172, 176
cdsd 11
cell (WebSphere) 44
Cell Directory Service, see CDS
certificate 37, 63, 116–117, 119, 123, 201
  client 40, 223
  from a CA 121, 252
  self-signed 120, 213
  subject DN 253
  test 124
  validity period 124
certificate authentication 40, 70, 73, 114, 117, 119, 195, 247
Certificate Authority (CA) 119, 121
certificates
  from a CA 252
channel binding 132, 135
cipher 123, 216
client certificate 40, 195, 223, 247
CN 245
common considerations 111
compound name 46
config.krb5 77, 166
configuration service 10, 56
conformant arrays 174
container 262
context handles 174
CORBA 23, 48, 63, 70, 174, 201
  COSNaming 195
  CosNaming 171
  CSIv2 195, 247
  GIOP 23
  IDL 23, 171, 174, 195
  IIOP 171, 195
  IIOP, see IIOP
  interoperability 73
  ORB 23, 175
  SDK 211

© Copyright IBM Corp. 2003. All rights reserved.

standards 23
WStringValue 235
corbaloc 47
corbaname 46–47
COSINE 29
CosNaming 24–25, 36, 45, 171, 185, 195
cred_handle 135
credential token 40
CRL 116
CSIv2 24–25, 37, 40, 70, 72–73, 195, 247–248
CustomRegistry 41

D
DAP 28
data streaming 174
DB2 115
DCE
cell 4
environment 11
KDC 11, 126, 128
login 126, 130, 153, 168, 179
registry 76, 94
review 3
dce_acl_* 56
dce_acl_inq_permset_for_creds 137
dce_db_* 55
dcecp 142, 151, 159–160, 245
dced 173
dcegroup 93, 97
dcegroupname 93, 97
DCERealm 140
dceXattrValue 107
declarative security 41
delegation 7, 38, 42, 56, 224
dependencies 12
direct 12
indirect 13
none 15
deployment phase 228
DES 169
DFS 4
digital certificate, see certificate
directory 5, 9
replacement strategy 58
Directory Access Protocol (DAP) 28
Directory Information Tree (DIT) 28, 114, 140
Directory Server (IBM), see IBM Directory Server
directory_jni 239
Distinguished Name, see DN
Distributed File System (DFS) 4
Distributed Time Service (DTS) 9
DMT 245
DN 28, 86, 94–95, 102, 140
Document Type Definition (DTD) 35
download scenario sample code 417
DTD 35
DTS 9

E
EAR 241, 244
ear file 224, 227, 229, 257–258
EIS 242
EJB 36
EJB container 44
EJB Security Collaborator 40
encryption 196
encryption cipher 123
enterprise bean 25, 35, 40, 70, 199, 248, 250, 254, 262
method 42
wrapper 201, 230
Enterprise Information Systems (EIS) 242
Enterprise JavaBeans (EJB) 36
ERA 8, 58, 76, 105–106
ERA entry 106
error handling and recovery 113
event management 10, 59
exceptions (IDL) 174
Extended Registry Attributes, see ERA
eXtensible Markup Language (XML) 33

F
form-based authentication 71
full pointer 174

G
GDA 9
General Inter-ORB Protocol (GIOP) 23
Generic Security Services API, see GSS-API
getCallerPrincipal 38, 42
getUserPrincipal 38
GINA 60
GIOP 23
Global Directory Agent (GDA) 9
Graphical Identification and Authentication (GINA)

60
GSKit, see IBM GSKit
gss_accept_sec_context 135
gss_acquire_cred 135
gss_init_sec_context 57, 135
gss_seal 133
gss_unseal 133
gss_unwrap 133
gss_wrap 133
GSS-API 5, 9, 53, 56, 62, 125–126, 152, 168
gssdce_* 57
gssdce_extract_creds_from_sec_context 137
gssdce_register_acceptor_identity 134, 155

H

High Performance File System (HPFS) 4
host management 10, 59
hostName 183
HPFS 4
HTTP 35, 71

I

IBM Directory Server 30, 117, 211, 252
  supported platforms 30
IBM GSKit 118
IBM Network Authentication Service 27, 75, 117, 125, 128, 138
  KDC server 53
  using with DCE data 75
IBM SecureWay Directory 30, 138
IBM Tivoli Access Manager 4, 22–23, 53, 85, 117, 125, 138
  using with DCE data 85
IBM TXSeries 4
IBM WebSphere 39, 117
  Application Assembly Tool 224, 255
  Application Server 13, 69, 175, 180, 201
  Authentication Module 40
  Authorization Engine 200
  cell 44
  Deployment Manager 44
  network deployment 44
  Security Collaborator 40, 200
  Studio Application Developer 211, 252
identity assertion 42
IDL 6, 24, 171, 174, 195, 204
idlc 189, 231
IETF 26, 36
IIOP 23, 39, 42, 70, 171, 195, 248
ikeyman 118, 213, 253
Implementation Repository 176, 182
indirect binding 176
inetOrgPerson 94, 99
InitializeSecurityContext 153
INS 47
integrated login 8, 59
Interface Definition Language, see IDL
Internet Engineering Task Force (IETF) 26, 36
Internet Inter-ORB Protocol, see IIOP
Interoperable Object References (IOR) 175
IOR 175, 199, 250
isCallerInRole 38, 42
isUserInRole 38

J

J2EE 25, 34, 37, 48, 70
  application environment 34
  standards 35
J2SE 36, 73, 238
JAAS 36, 40
JAF 36
jar file 223
Java 34
  applet 35, 38, 74
  authentication 37
  authorization 37
  enterprise bean 35
  JDK 211, 252
  JSPs 35
  RPC 38
  servlets 35
  threads 39
  URL names 251
Java 2 Standard Edition (J2SE) 36, 73
Java API for XML Parsing (JAXP) 36
Java Authentication and Authorization Service (JAAS) 36
Java Connector Architecture (JCA) 35–36, 74
Java Database Connectivity (JDBC) 36
Java Message Service (JMS) 36
Java Naming and Directory Interface, see JNDI
Java Native Interface (JNI) 73
Java Remote Method Invocation (RMI) 70
Java Server Pages (JSP) 25, 36
Java Servlet Specification 36
Java Transaction API (JTA) 36

JavaBeans Activation Framework (JAF) 36
javah 239
JavaMail 36
JAXP 36
JCA 35–36, 74, 241
JCA connector 198
JCA container 44
JDBC 36, 38
JDK 211, 252
jdouble 241
JMS 36
JNDI 13, 30, 36, 38, 45, 70, 171, 195, 247
JNI 73, 198, 238
JNIConnection 242
JSP 36, 44, 224
JTA 36
JVM 238, 250

K

kadmin 149, 165–166
kadmin.local 162, 166
kadmind 166
kdb5_util 166
KDC 11, 53, 126, 128, 139, 165
kdestroy 166
Kerberos 26, 48, 125, 263
key database 120
key file 120
keytab 155
kinit 166
klist 166
kpasswd 167
krb5.conf 77
krb5_util 166
krb5kdc 165
krbdeletetype 101, 151, 163
krbKdcServiceObject 77, 79, 141
krbpolicy 103
krbprincipal 92, 96
krbprincipalname 92, 96
krbPrincSubtree 77, 93, 141
krbprolicyname 103
krbRealm 140
krbRealmName-v2 140
KrbRealm-v2 140
krbTrustedAdmObject 77, 79, 141
ksetup 150, 167
ktutil 167

L

launchClient 259
LDAP 28, 41, 49, 76, 85
  ERA data 105
  replication 115
  standards 29
LDAP directory 20, 28, 58, 75, 86
  availability 115
  migrating DCE data 76
  security 114
  using and considerations 114
LDAP filter rules
  scenario 3 218
  scenario 4 253
ldapadd 140
ldapmodify 161
LDAPS 157, 167
LDIF 140
Lightweight Directory Access Protocol, see LDAP
Lightweight Third Party Authentication, see LTPA
Linux 48
LocalOS 41
logging 31, 49
login 7, 37, 126
  integrated 8
LTPA 40, 72, 195, 247

M

master key 77
messaging 38, 60
messaging service 10
Microsoft Internet Explorer 114
Microsoft Platform SDK 138
multiple-master replication (LDAP) 115
multi-threaded 175
mutex 68

N

Naming Service (OMG) 24
National Security Assotiation (NSA) 31
Netscape 36, 114, 140
Network Authentication Service
  see IBM Network Authentication Service
Network Time Protocol, see NTP
NTP 30, 49, 69

O
OASIS 34
Object Request Broker (ORB) 23
OMG 23, 36
OMG Naming Service 24
Open Management Group, see OMG
Open Software Foundation 4
OpenSSL 122
ORB 23, 25, 175
OS/390 27, 48
OS/400 27, 48

P
PAC 7, 38, 53, 72, 126, 247
packet privacy 196
PAM 60
Parallel System Support Program (PSSP) 4
password strength 8, 61
PD.Acld 138
PD.AuthADK 138
PD.Mgr 138
PD.RTE 138
pdadmin 165, 167
performance considerations 113
personal certificate (see also certificate) 118
PGO entry 106
Platform SDK (Microsoft) 138
Pluggable Authentication Module (PAM) 60
policy 159
Policy Director 4, 85
POP 159
POSIX 32
POSIX 1003.c 67
preface xix
priv_* 56
private key 117, 123
Privilege Attribute Certificate, see PAC 6
programmatic security 41
Protected Object Policies (POP) 159
protection 38, 62, 126
pthreads 32, 67
public certificate, see certificate
Public Key Infrastructure (PKI) 117

Q
QoS 193

R
rdacl 54, 136
RDN 28, 140
realm 76
realm entry 140
Redbooks Web site 423
  contact us xxii
  downloads 417
regimpl 176, 192
RegisterEventSource 32
registry 8, 41, 62
Relative Distinguished Name (RDN) 28
remote cache mode 152
remote procedure call (RPC) 5
replacement considerations 111
replacement roadmap
  scenario 1 138
  scenario 2 180
  scenario 3 210
  scenario 4 251
replacement scenarios
  scenario 1 125
  scenario 2 171
  scenario 3 195
  scenario 4 247
replacement strategies 51
  auditing 52
  authentication 53
  authorization, PAC, UUID 53
  backing store 56
  C/C++ applications 52
  configuration 56
  delegation, GSS-API, login 56
  directory 58
  event management 59
  extended registry attributes (ERA) 58
  host management 59
  integrated login 59
  Java applications 69
  messaging 60
  mixed applications 73
  password strength 61
  registry 62
  RPC services 63
  serviceability 66
  threads 67
  time 69
replacement technologies 19
ReportEvent 32

Resource Adapter 242
rgy_edit 159, 203, 245
RMI 42, 70, 248
RMI over IIOP 247
rmic 231
RMI-IIOP 38
role-based authorization 195, 247
root certificate 119, 121, 123
root certificate (see also certificate) 118
root context 177
RPC 38, 53, 136, 196
RPC, see remote procedure call
rpc_*auth* 65
rpc_binding_*auth_* 56
rpc_ep_* 65, 178
rpc_ep_register 179
rpc_mgmt_ep_* 65
rpc_network_is_protseq_valid 179
rpc_ns_* 65, 178
rpc_ns_binding_export 179
rpc_ns_binding_import_begin 178
rpc_ns_binding_import_done 178
rpc_ns_binding_import_next 178
rpc_server_* 178
rpc_server_inq_bindings 179
rpc_server_listen 179
rpc_server_register_if 179
rpc_server_use_protseq 179
run-as 42, 263

S

SAS 40
sas.client.props 200, 250
scenario 1 125
scenario 2 171
scenario 3 195
scenario 4 247
SDK (CORBA) 211
sealed delegation 42
sec_acl_* 56
sec_attr_base.h 106
sec_cred_* 56
sec_ctx 135
sec_id_* 56
sec_login_* 57
secret key 117
Security Collaborator (WebSphere) 40
security considerations 113
security role mapping 41
Security Service Provider Interface (SSPI) 152
self-signed certificate 120
server alias 182
serviceability service 10, 66
servlet 25, 71, 74, 224
    mappings 224
signer certificate (see also certificate) 118
Simple Network Time Protocol (SNTP) 30
Simple Object Access Protocol (SOAP) 33
simplifications in scenarios 112
Single SignOn 220
single-master replication (LDAP) 115
SNTP 30
SOAP 33
sockets 129
Solaris 48
source code listings
    scenario 1 267
    scenario 2 331
    scenario 3 363
    scenario 4 403
SQL 38
SSL 24, 36–37, 70, 72, 114, 116, 195, 247–248
SSL implementation hints 116
SSL key files
    scenario 3 213
    scenario 4 252
SSL, uses of 117
SSO 220
SSPI 153, 168
standards
    for CORBA 23
    for J2EE 35
    for LDAP 29
    for SSL and TLS 116
    for threads 32
    for Web services 33
start.krb5 167
stateless session bean 230, 248
stop.krb5 167
streaming 174
strong authentication 117
subject 119
svrsslcfg 153
SWAM 40
syslog 31

T

technologies
  for C/C++ applications 21
  for Java applications 34
test certificate 124
TGT 165
The Open Group 4, 22, 31, 55
threads 5, 32, 39, 67, 175
ticket refreshing 113
time 69
Tivoli Policy Director 4, 85
TLS 24, 36, 116
trust file 120

U

UDDI 33
uddi.org 33
unconfig.krb5 167
Universal Description Discovery and Integration (UDDI) 33
usage binding 184
UUID 10, 26, 48, 53, 55, 69, 76, 177

V

validity period (certificate) 124
VeriSign 122
Visual Age for C++ 211
Visual C++ 211
VtlUtil 235

W

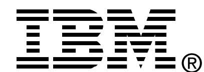
W3C 33
war file 224
WASPROPS 222
Web container 44
Web Portal Manager 149, 167
Web services 33, 49, 64, 263
Web services Description Language (WSDL) 33
WebSEAL 23
WebSphere Administrative Console 245, 258
WebSphere, see IBM WebSphere
Windows 48, 126, 128, 149–150
wrapper (enterprise bean) 230
wsadmin 245
WStringValue 235

X

X.500 29
X.509 117
  see also certificate
XML 25, 33–34
xntpd 69
  daemon 31
XPG4 32, 60

DCE Replacement Strategies

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages



DCE Replacement Strategies

Redbooks

Replacement technologies

Replacement strategies

Best practice scenarios and coding examples

This IBM Redbook recommends strategies that you can use to replace the Distributed Computing Environment (DCE) dependencies in your environment and move to new technologies. The following topics are covered:
- DCE overview and recap
- Replacement technologies
- Replacement strategies
- Replacement scenarios
- Replacement coding examples

This book is a valuable information source if you are an executive, administrator, or developer of an IBM customer environment that uses IBM DCE for a distributed systems and application infrastructure.

Although strategies for replacing DCE are described, the book does not cover strategies for replacing dependencies to IBM products that use DCE, such as DFS and TXSeries.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-6935-00 ISBN 0738453080