E-DIRECTORIES
Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and we were aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of IBM. Printed in the United States of America. Published simultaneously in Canada.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact

Pearson Education Corporate Sales Division
One Lake Street
Upper Saddle River, NJ 07458
(800) 382-3419
corpsales@pearsontechgroup.com

Visit AW on the Web: www.awl.com/cseng/

Library of Congress Cataloging-in-Publication Data
e-Directories : enterprise software, solutions, and services / Daniel House ... [et al.]
p.m.
   Includes bibliographical references and index.
   I. House, Daniel.

TK5105.595.E18 2000
005.7—dc21

Copyright © 2000 by International Business Machines Corporation


Text printed on recycled paper
1 2 3 4 5 6 7 8 9 10—CRS—04 03 02 01 00
First printing, July 2000
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>xxi</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>xxvii</td>
</tr>
<tr>
<td>About the Authors</td>
<td>xxix</td>
</tr>
<tr>
<td>Introduction</td>
<td>xxxi</td>
</tr>
</tbody>
</table>

## Part 1: You Are Here

### Chapter 1: Introduction to the Problem

- **A User-Oriented View**
  - Over and Over
  - Data Sharing and Manipulation
- **A System View**
  - Location, Location, Location
  - Administering to the Masses
  - A Unifying Force
  - Identity
  - Enterprise Characteristics
- Bringing the Views Together with LDAP
  - Lightweight Directory Access Protocol
- What, Exactly, Is a Directory?
- Where to Find Code
- Summary

### Chapter 2: What Directories Are and Are Not

- Directory Gestalt
- Server and Service
- The Importance of RASSS
- What Directories Do
  - Functional Characteristics of a Directory Service
<table>
<thead>
<tr>
<th>Chapter 3</th>
<th>Directories Are Everywhere</th>
</tr>
</thead>
<tbody>
<tr>
<td>What a Directory Is Not</td>
<td>16</td>
</tr>
<tr>
<td>Remember ACID?</td>
<td>16</td>
</tr>
<tr>
<td>Not Quite Nirvana</td>
<td>17</td>
</tr>
<tr>
<td>Loose Consistency</td>
<td>17</td>
</tr>
<tr>
<td>A Definition Can't Be Wrong—by Definition</td>
<td>18</td>
</tr>
<tr>
<td>Good, Bad, and Ugly Data</td>
<td>18</td>
</tr>
<tr>
<td>A Brief Code Example</td>
<td>20</td>
</tr>
<tr>
<td>Summary</td>
<td>21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4</th>
<th>Directories: Bone Yard to Nursery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early Standardization</td>
<td>37</td>
</tr>
<tr>
<td>SNA Host Names and Logical Units</td>
<td>37</td>
</tr>
<tr>
<td>Domain Name Service</td>
<td>37</td>
</tr>
<tr>
<td>Distributed Computing Environment Cell Directory Services</td>
<td>40</td>
</tr>
<tr>
<td>LAN Managers</td>
<td>41</td>
</tr>
<tr>
<td>X.500</td>
<td>42</td>
</tr>
<tr>
<td>Standards Today</td>
<td>43</td>
</tr>
<tr>
<td>X.500</td>
<td>44</td>
</tr>
<tr>
<td>Domain Name Service</td>
<td>44</td>
</tr>
<tr>
<td>Lightweight Directory Access Protocol</td>
<td>46</td>
</tr>
<tr>
<td>Summary</td>
<td>48</td>
</tr>
</tbody>
</table>
Part 2 Enterprise Software with Directories 51

Chapter 5 Man-Rated, Enterprise-Rated, Good-Enough, and Don’t-Care 53
- Four Classes of Usage 53
- How Much Software Is Written and at What Cost? 54
- Examples of the Four Usage Classes 55
- Cost 56
- Directory Exploitation 58
- A One-Line Quiz 59
- Example Program 61
- Summary 65

Chapter 6 The Operating System and the Directory 67
- The Role of the Operating System 67
- Usage Ratings 68
  - Usage Rating of the Operating System 68
  - Usage Rating of the Directory Service 68
  - Directory Service Users 69
  - Compensating for Low Usage Rating 70
- Fault Tolerance, Redundancy, and Graceful Fail-Over with Takeover 70
- Summary 73

Chapter 7 Directory Users 75
- People 75
  - White Pages and Yellow Pages Lookups 75
- Operating Systems 77
  - Users and Groups 77
  - Hardware Configuration 78
- Networks 80
- Applications 81
  - Configuration 82
  - Location 82
  - Users and Groups 83
- Summary 84

Chapter 8 The Evolution of Application Models 85
- Enterprise Application Models 85
  - Monolithic Applications 85
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Client-Server Applications</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>Thick Client Applications</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>Thin Client Applications</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>Tiered Applications</td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>Client-Server Glue Technologies</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>Transactions</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>Message-Oriented Middleware</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>Boss-Worker Thread Model</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>Managed Components</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>Applications on Symmetric Multiple Processors</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>Clustering</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>Workload Balancing</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>Fail-Over Clusters</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>94</td>
</tr>
<tr>
<td>9</td>
<td>Chapter 9: Availability</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>24 × 7 and 24 × 365</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td>Servers and Services</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>High Availability</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>Robust Server Code</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>Redundant Servers</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>Low Availability</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Improving Application Availability</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>Advertising and Finding Services</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>Beyond Location Information</td>
<td>105</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>106</td>
</tr>
<tr>
<td>10</td>
<td>Chapter 10: Scaling</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>Scalability</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>Planning Ahead</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>Approaches to Scaling</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>Single-System Scaling</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>Multiple-System Scaling</td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>Directories and Scalability</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>115</td>
</tr>
<tr>
<td>11</td>
<td>Chapter 11: Performance</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>Performance and Performance Measurement</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>Gating Factors</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td>Processor Usage</td>
<td>118</td>
</tr>
</tbody>
</table>
Contents

Input/Output Usage and Constraints 120
Network Bandwidth, Latency, and Timing Fluctuations 121
Poor Performance 122
Summary 123

Chapter 12 Security 125
Policy and Risks 125
Application Development and Maintenance as a Source of Ongoing Risk 127
Controlling Access Based on Organization Responsibilities 128
The Audit 131
Summary 132

Chapter 13 Life Cycle Control 133
Serviceability and Maintainability 133
Controlling Downtime 134
Real-Time Event Logging with First Failure Data Capture 134
Failing a Request Versus Failing the Server Application 136
Directory Redundancy 139
Periodic Maintenance 140
Backups and Directories 140
Planning for the Next Release 141
Carrying Out an Upgrade 141
Installing a New Server in Place 142
Streaming Data to a File 143
Applying the Life Cycle Lessons 143
Client and Server Changes 144
Availability 145
Summary 145

Chapter 14 Directories Grow Up 147
Predicting the Future 148
Integration with the Operating System 148
Transactional “Flavors” 148
Storing Large Objects 150
Dynamic and Easy Partitioning 150
Hot Spot Management Through Dynamic Replica Creation 151
Better Multivendor Interoperability 151
XML and DSML 152
Improved Administration 153
Monitoring Tools 153
Chapter 19  Replication and Partitioning  225

Replication
  How Replication Works
  Application-Level Problems Caused by Replication
  Replication Versus Caching
  Dealing with Temporal Consistency

Partitioning
  Partitioning for Performance
  Partitioning for Management
  Application-Level Problems Caused by Partitioning
  Referrals
  Drawbacks of Referrals
  Chaining
  Putting Replication and Partitioning Together
  Business Issues Drive Replication and Partitioning
  Summary

Chapter 20  Synchronization and Metadirectories  241

Replication
  Synchronization
    Access Control Problems
    Schema Mapping Problems
    Replication Problems

Metadirectories
  Summary

Chapter 21  APIs and Protocols  253

Differences between APIs and Protocols
  Protocol
    Programming Interface
      Origins of Directory Protocols and Programming Interfaces
        Novell Directory Service (NDS) and Network Core Protocols (NCPs)
        Distributed Computing Environment (DCE) and Cell
        Directory Services (CDS)
        X.500 and Directory Access Protocol (DAP)

Popular Directory Protocols
  LDAP
  Domain Name Service (DNS)
  Novell Directory Access Protocol
  COSNaming over IIOP
Popular Directory Programming Interfaces 261
  LDAP C 261
  JDAP 262
  PerLDAP 263
  Novell Development Kit (NDK) 263
  COSNaming 265
  JNDI 265
  Active Directory Service Interface (ADSI) 266
Summary 267

Chapter 22 Directory Implementations 269
  Commercial Implementations 269
    IBM SecureWay Directory 269
    Novell Directory Services (NDS) 269
    Netscape Directory Server 270
    Microsoft Active Directory 270
    Lotus Domino Name and Address Book 271
    Banyan Vines 271
  An Open Source Implementation 271
  Specialized Implementations 272
    Domain Name Service (DNS) 272
    Service Location Protocol (SLP) 273
Summary 273

Part 4 Internet and Intranet Case Studies 275
Chapter 23 Roaming Users with LDAP 277
  How Directories Enable Roaming 277
  Description of the Roaming Problem 278
  The Sales Application (TSA) 280
  Schema 281
    TSA Schema Class Objects 281
    TSA Schema Attributes 282
    The Directory Information Table 283
    Updating the Schema 285
    Directory Vendor Differences 285
    Other Schema Issues 286
  Directory Structure 286
  How TSA Works 287
    Write Activity 289
    Write Collisions 289
    What TSA Looks Like 290
Chapter 24  Corporate Employee Directory  293
The Purpose of a Corporate Employee Directory  293
Use Cases for the Corporate Directory  294
Data Model  295
Object Classes  295
Attribute Types  296
Name Space Layout  297
Accessing the Corporate Directory  299
Looking Up People  299
Looking Up a Chain of Command  300
Adding a Distribution List  301
Modifying a Distribution List  302
Deleting a Distribution List  302
Managing Information in the Corporate Directory  303
Adding Employee Information  304
Modifying Employee Information  304
Deleting Employee Information  304
Handling Employee Transfers  305
Handling Company Reorganization  308
Sample Code  308
Using the Corporate Directory from E-Mail Systems  309
Critique of the Solution  309
Summary  310

Chapter 25  Personalization of the Internet  311
The Customer Access and Update  311
The Web Self-Help (WSH) Site  312
Schema  312
Directory Infrastructure  313
How the Web Self-Help Application Works  315
Critique of the Solution  318
Summary  319
Chapter 26  Application Management 321
  The Application Management Problem 321
  The Purpose of the Application Management 322
  The Directory 323
  Directory-Enabling the Server 324
  Directory-Enabling the Client 326
  Schema 327
  Critique of the Solution 328
    Reliability 330
    Availability 330
    Serviceability 331
    Scalability 331
    Security 331
  Summary 332

Chapter 27  Internet and Intranet Single-Sign 333
  Internet and Intranet Users 333
  The Single Sign-On Solution 335
  The Directory Solution 336
  Summary 337

Chapter 28  Configuration Management of a Web Farm 339
  The Web Farm Problem 339
    Web Farm Design 340
   Adding Servers 343
    Managing the Web Farm 344
    Implementing the Web Farm 344
  Management Objectives 347
    Centralized Control of the Constituent Servers 348
    Transparent Replication of Content Updates 349
    Web Service Advertising: Mapping of DNS to Correct IP Addresses 349
    Performance Monitoring 349
    Crisis Notifications 350
    Hot-Plugging and Removing Servers 350
    Backup and Restoration of Backing Store Without Service Interruption 350
    Migrating Individual Boxes 351
  Schema 351
    The Directory Infrastructure 351
    Directory Configuration 352
Appendix C  ADSI  
Using ADSI  395
ADSI and C/C++  395
For More Information and Samples  396

Appendix D  Web APIs and Protocols  
Projecting Static Brochures to Clients  397
Web Protocols  399
Web Client APIs  400
Client-Side Scripting  400
Java Applets  401
ActiveX Controls  402
Dynamic HTML with Cascading Style Sheets  403
Web Server APIs  404
Common Gateway Interface  404
Improving CGI  405
Session State  406
Transactions  408
Server-Side Components  408
Security  411

Appendix E  Generic Security Services (GSS-API) and System Security Provider Interfaces (SSPI)  
General Helper Functions  414
Mechanism/Package Management  414
Buffer Management  414
Name Management Functions  415
Object Identifier Helper Functions  415
Other Functions  416
Credential Management Functions  416
Context Management Functions  417
Message Protection Functions  420
**Appendix F  Access Control Summaries** 421
- IBM SecureWay Directory Access Control 421
  - Users and Groups in Access Control Lists 421
  - Structure of an Access Control List 422
  - Evaluation Algorithm for an Access Check 423
  - Creating a New Object 423
  - Common Problems and Solutions 423
- Microsoft Active Directory Access Control 424
  - Users and Groups in Access Control Lists 424
  - Structure of an Access Control List 424
  - Evaluation Algorithm for an Access Check 426
  - Creating a New Object 426
  - Common Problems and Solutions 426
- Netscape Directory Access Control 427
  - Users and Groups in Access Control Lists 427
  - Structure of an Access Control List 427
  - Evaluation Algorithm for an Access Check 429
  - Creating a New Object 429
  - Common Problems and Solutions 429

**Appendix G  IBM Standard Schema** 431
- Standard Directory Information Tree (DIT) 431
  - Standard Objects 431
  - Attributes 432
  - Syntaxes 432

**Appendix H  Directory Standards and Other References** 435
- IETF RFCs 435
- IETF Drafts 440
- Other IETF Documents 443
- Other Standards 444
- Vendor Information 444

**Appendix I  Useful Attribute Types and Object Classes** 447

**Appendix J  Using the CD** 461
- CD Contents 461
- Using the CD 462
  - Startup.exe 463
  - Browser Versions, Executable Programs, and File Formats 463
  - Executable Programs and Your Operating System 463
Intended Audience

This book is for information system architects, enterprise software developers (or those who want to be), consultants, students, and technology managers who need to understand the architecture, benefits, and services provided by directories. The goal is to provide the reader with background, motivation, and details on building enterprise software solutions using directory services. The book is not about installing and configuring directory services, although there is information on those topics on the CD. Rather, this book is about Internet scale software and how it can use directory services to solve business problems.

What Is a Directory?

A directory is a distributed data storage and retrieval mechanism. It has some of the characteristics of a database, a transaction processing system, and a file system. A directory is a wonderful place in which to manage information so that multiple operating systems, or multiple instances of one operating system, can use it. Directories can hold nearly anything: descriptions of people and their e-mail addresses, locations of vital services in the network, configuration data, user-defined data, and more.

What Is Enterprise Software?

Enterprise software is distinguished from other software in the critical nature of its function. It is commonly the application or set of applications that run an enterprise, whether it’s a company, school, government agency, or international organization. Directories are both enterprise software and enabling technology for enterprise software.

As enterprise software, directories are a key part of a software system that must exhibit the highest levels of performance, scalability, and robustness. As enabling
technology for enterprise software, directories enable other applications to be written that themselves must exhibit the highest levels of these characteristics.

**What Is LDAP?**

The Lightweight Directory Access Protocol (LDAP) is a relatively new standard for accessing directory services. This exciting new protocol and its API are being implemented across a wide range of directories and other products. While this book is not solely about LDAP, many of our examples use it because it is such an important new standard.

**What if You Prefer Java?**

For the enterprise Java engineer or professional, in addition to C-based LDAP programming, we cover Java Naming and Directory Interfaces (JNDI). Many of our examples use C or C++ with LDAP, but some are provided in both LDAP C and JNDI. A very few are provided only in JNDI.

For good measure, and to show you that there are other ways to program directories, a small number of examples use still other APIs, such as PerLDAP (a close match for LDAP in Perl) and ADSI (Microsoft’s COM-based Active Directory Services Interfaces).

**What if You Don’t Know LDAP or JNDI?**

If you have no LDAP familiarity at all, the best way to use this book is to read Part 1 and browse Appendix A as needed (you should read Appendices A and B before starting Part 2). Appendix A is a quick summary of LDAP from a programming perspective, and Appendix B is a quick summary of JNDI. If you need more complete LDAP and JNDI API references, you will find them on the companion CD in the Redbooks section (for LDAP) and the JNDI section.

**What This Book Is and Is Not**

This book explores enterprise class software and how directory services can be exploited to improve the enterprise characteristics of your software. Enterprise is a
word we use to denote “important” software, such as business critical systems, often implemented in heterogeneous operating system environments. Enterprise characteristics are those qualities that make software suitable for business critical systems.

This book is not an API reference (although LDAP and JNDI references and other sources can be found on the companion CD). Instead of walking you through the parameters of an API set, our goal is to build you a mental toolkit of enterprise software techniques with directories. With these techniques and an enterprise mindset, you will be able to build more intelligent, larger scale, and more robust systems using directories.

Example Programs

It is difficult to provide examples for real directory services. There are many vendors, many differences among implementations and nuances of supported features, and a relatively low likelihood that you already have a directory server running on your laptop or desktop.

Some of the example programs provided with this book were built using Microsoft’s Windows 2000 Server and its built-in Active Directory. Other examples were built using IBM’s OS/390 and the IBM SecureWay LDAP Directory or with Linux and the OpenLDAP Directory Server. To build and run any of these yourself, you would need to make some changes due to your configuration, chosen directory service, schema, or other factors.

All of the example programs could be run with modification using the IBM SecureWay LDAP Directory, which is included on your CD, or another LDAP directory. Most examples will need at least minor changes for your environment, and some examples may need major changes (for example, porting a Windows GUI application to X-Windows). For instance, you might need to install the Java JDK and JNDI for JNDI examples (a JNDI client is supplied on the CD), to install an LDAP C client (again, one is supplied on the CD), to supply appropriate security credentials, or to supply different directory entry names or directory server names.

Each example program is provided on the CD with complete source code, a summary describing its features, and hints on how to port it to other platforms or other directory services.

Keep in mind that any example programs shipped with this book are just that: examples of techniques. It is not our intent to cover every possible error case, even though enterprise software would do that. Most of the example programs were written specifically to illustrate a few ideas and are not from a production system. You might even find bugs.
If you do find bugs or other interesting “features” in any of our example programs, we would like to hear from you. You can contact any or all of us by e-mail using the addresses in About the Authors. Sorry, there is no “bug bounty.”

The Companion CD

The companion CD contains many interesting and useful things.

- This book! So you can take your laptop on a business trip and pack the CD instead of the printed version, if that is your preference.
- Example programs. All of the sample programs are contained on the CD and organized in a chapter/appendix directory tree easily reachable from a root HTML page.
- Complete working versions of the IBM SecureWay LDAP Directory Server for Microsoft Windows and other supporting products. You can install these from the CD, but you must agree to the license terms, which state that this software can be used only for development purposes—production use, among other things, is strictly prohibited.
- C LDAP client software developer kits (SDKs) for Microsoft Windows, Sun Solaris, and IBM AIX.
- A multiplatform JNDI client SDK. If you want to program in Java instead of using the C-based LDAP client SDKs, this is for you.
- Complete versions of useful Redbooks from IBM Engineers. 
  Understanding LDAP, which contains LDAP API information, programming samples, and other background, SG24-4986-00
  LDAP Implementation Cookbook, which has information on planning for a directory service, installation, configuration, and some examples, SG24-5110-00
  Ready for e-business: OS/390 Security Server Enhancements, which describes LDAP uses on OS/390, SG24-5158-00
  Enterprise JavaBeans Development Using VisualAge for Java, Chapter 6, which contains information on JNDI, SG24-5429

- Other technical books containing useful information. The following books, while specific to the OS/390 platform, contain information that is applicable to other platforms as well.
  OS/390 Security Server: LDAP Server Administration and Usage Guide, has information on running a particular LDAP server for large systems, SC24-5861-04
- The complete Java Naming and Directory Interfaces (JNDI) specification from Sun Microsystems. It’s fully hyperlinked and ready to browse or use as a reference.
- The IBM Common Schema. These are the schema objects IBM products use to extend multiple directory servers. This snapshot package (schemas do change) is fully hyperlinked and ready to browse.
- Other supplementary write-ups and materials. See the welcome.htm page in the CD root for links to other information.

All of this material can be reached from a home page on the CD (welcome.htm), or by navigating the directories on the CD. The directory structure of the CD is intentionally very simple to allow easy exploration.

**The Organization of This Book**

This book is divided into five parts. The descriptions and aims of each section are as follows.

- Part 1 introduces the problem, including the origins of directories, the fundamental problems, and the history behind directory services.
- Part 2 describes how enterprise software characteristics relate to directory services. Chapters cover availability, security, and usage categories of directory services. This section shows how directories can be used to improve the enterprise characteristics of your software.
- Part 3 contains some nuts and bolts of directory services, with chapters on directory replication and partitioning, schemas, and descriptions of some directory products. In addition to the basics of directory services construction, this section offers a glimpse of where directory services may be going in the future.
- Part 4 consists of case studies. Chapters cover specific examples of how directory services can be used to solve real problems, such as personalizing Internet access and managing applications through common profiles.
- Part 5 is the reference section with URLs, API and protocol descriptions, RFCs, the glossary, and other information. Appendix H, for example, contains URLs for numerous topics (on the CD the links are all live and can take you to the latest vendor information, RFCs, and other places). Appendices A, B, and C are API mini-references. Most of the things in this part are meant to be referenced when needed, rather than read sequentially.
How to Use This Book

How you use this book will depend on your experience. For example, if you already know a great deal about enterprise software and about directories, you may find the case studies and reference material interesting, but perhaps not the earlier sections. Here is a rough guide.

If You Already Know a Great Deal About:  Then You Will Want:
Neither directories nor enterprise computing  The entire book
Enterprise computing  Parts 1, 3, 4, and 5
Directories  Parts 2, 4, and 5
Enterprise computing and directories  Parts 4 and 5

We hope there is something here for everyone who wants to learn about directories and how they can enable enterprise class computing. Enjoy!
We would like to thank Mike Schlosser for starting the LDAP implementation cookbook and for providing our Metadirectory case study (Chapter 29). Thanks to Ellen Stokes for her helpful advice, and to James Mannon, Jamil Bissar, and Chris Pascoe for their help in getting the SecureWay LDAP Directory on the CD. In addition we wish to thank David Goodman at Lotus for his article, “The Domino Directory: The Foundation of a Directory-Enabled Infrastructure,” and to Jeffrey Snover at Tivoli; Heinz Johner and Ian Crane of the IBM ITSO Redbook organization; and Pat Gibney, Dick Sullivan, and Bill Behan for their help and sponsorship. Our thanks also to the others at IBM who helped: Christina Lau, Greg Doudnikoff, Mark Simpson, Dave Peterson, Dave Dyar, and Pat Fleming.

Many reviewers helped improve early drafts. We would like to thank the ones who don’t mind being mentioned publicly for their invaluable help: Rosanne Lee, James Pinpin, Richard Brooks, James Frentress, Anton Stiglic, and Maynard Johnson.

Of course the great staff at Addison Wesley deserves much of the credit. They include Julie DiNicola and many more. We would also like to thank the production team: Kim Arney, the project manager and compositor; Carol Noble, the copyeditor; Pat Menard, the proofreader; and Jason Jones and Katie Noyes, the CD designers. Finally, we want to thank our families for putting up with this project: Susan, Matthew, and Graham House; Carey Mauget; Jeannie, Megan, and Allen Daugherty; and Jeanne, Jeffrey, and Dylan Hahn.
ABOUT THE AUTHORS

Daniel E. House  Dan is a Senior Technical Staff Member at IBM in Research Triangle Park, North Carolina. He has worked for IBM in diverse areas such as large systems hardware and software, distributed computing, transaction processing, application development tools, operating system design, and Microsoft Windows technologies. He holds a bachelor of arts degree in computer science from Indiana University, Bloomington, Indiana, and a master of science degree in systems and information science (computer science and computer engineering) from Syracuse University, Syracuse, New York. He holds seven U. S. patents, with about a dozen more pending. Dan is a member of the Association for Computing Machinery (ACM) and is a senior member of the Institute of Electrical and Electronics Engineers (IEEE). Dan can be reached at dhouse@us.ibm.com, or d.house@computer.org.

Timothy Hahn  Tim is a Senior Technical Staff Member at IBM. He is responsible for strategy, architecture, design, and development of directory services on OS/390. Tim joined IBM in 1990, working on communications processes in the IBM Virtual Machine/Control Program (VM/CP). He has worked on many projects within IBM, including Distributed Computing Environment (DCE) services, security services, and, most recently, Directory Services and LDAP. He has experience in the design and development of database applications as well as communications, security, and directory services in highly scalable environments. He holds a bachelor of science degree in electrical engineering from the University of Notre Dame and a master of science degree in electrical engineering from the University of California at Berkeley. Tim can be reached at hahnt@us.ibm.com, or thahn@stny.rr.com.

Louis Mauget  Lou is a Senior Software Engineer in the Windows 2000 System Manager Group at IBM in Research Triangle Park, North Carolina. He has worked as a programmer and developer in numerous areas during his 34 years at IBM. Lou’s experience includes on-site enterprise customer services, operations research, prototyping applications of new software technologies, business application development, embedded monitoring and control applications, handheld device applications, product development, and numerous technology assignments. He holds a bachelor of science degree in physics from Michigan State University. Lou can be reached at mauget@us.ibm.com.
Richard Daugherty  Rich is a Senior Software Engineer in the Windows 2000 System Manager Group at IBM in Research Triangle Park, North Carolina. He has worked as a developer, architect, and security specialist in various product groups within IBM, including large system operating systems, networking architecture, and the Windows NT/2000 system manager group. Rich also spent two years in the IBM Security Consulting Practice, leading such security projects as Enterprise Internet Policy and Intrusion Testing. He holds a bachelor of arts degree in mathematics and computer science from Western Kentucky University and a master of science degree in computer science/software development from Marist College, Poughkeepsie, New York. Rich can be reached at richd1@us.ibm.com.
INTRODUCTION

The Scope of Directories
At the risk of sounding trite, a directory is serious business software that enables other serious business software. The world has become networked, and directories are a tool to bring the utmost in reliability and robustness to distributed enterprise software across intranets and the Internet. Directories can be used in many ways, one of which is to enable disparate operating systems to share a common view of vital data (as such, they are wonderful “heterogeneous OS glue”).

Enterprise Software and Directories
Enterprise software is more than just a collection of database, transaction processing, communications, and operating systems. It is the software that holds an organization, or even the modern world economy, together. It is a set of principles that guide important software application design and development, and the infrastructure on which important applications are built. Badly written enterprise software causes service outages, angry customers, lost revenue, lost jobs, and possibly even lost lives. Well-written enterprise software is invisible: Everything just works.

There are innumerable pieces of software infrastructure that could be studied. Numerous books and papers have been written on software design and on how to use a particular set of software interfaces. This book builds a story, with examples, about creating “industrial-strength” distributed software applications using the class of enabling software called directories.

Have You Ever Used a Directory?
Most people have never used a directory. Or so they think. Actually, everyone has used a directory and chances are good that you have used dozens of them. They usually work invisibly, behind the scenes in large organizations or Web sites.
Directories are much more important than is generally realized. Most serious, large, complex software systems use a directory, whether or not it is called a directory. Recently, directories have begun to emerge as key enterprise-enabling technologies in themselves, and more applications are being written that use them (and expect them to behave well).

As directories have emerged from behind the scenes, they have added more functions. They have become much more robust and capable of handling larger quantities of data and rates of data exchange. They have also become more standardized and easier to exploit. Finally, they have become vital to the increasingly important Internet. So why aren’t there more books on building enterprise-class software with directories? We don’t know, but we decided to write this one.

In this book, we use the key emerging technology of standardized directories to illustrate the characteristics of enterprise software. We use directory technology in examples to show how this key enabling technology can be the foundation for large-scale, multiplatform, Internet- or intranet-based, mission-critical applications.

We use several cross-platform and platform-specific enabling services to build example applications. JNDI and LDAP are both cross-platform enabling services. LDAP is a generic interface to directory services, sanctioned by the Internet Engineering Task Force (IETF) and implemented in directory servers by many vendors. As such, it is a standard, powerful, cross-platform, “plumbing” service and set of APIs (we’ll clean up the inaccuracies in that statement later, even though they are not of much practical consequence). JNDI is one way to access LDAP “on the wire.”

**Attitude Versus API**

Enterprise computing is more about attitude than it is about specific tricks and techniques or APIs. A full mental toolkit of technology tricks helps, but a mental state that is concerned with enterprise principles is trick number one. Scalability, throughput, response time, availability, serviceability, security, monitoring, and ease of use are principles that, when respected, will guarantee that everything just works, even on the busiest business day of the year. Directory technologies, combined with enterprise principles and attitude, make for a very powerful skill combination that will make your software solutions better and ready for the networked world.

The rest of this book provides you with a mental toolkit of techniques for enterprise software with directories and gives you the enterprise attitude as well.
The Top 10 Terms and Acronyms

There is a glossary in the back of this book, and many terms are defined throughout the book as needed. However, there are a few terms that are commonly misunderstood, or are overloaded from other definitions, or are so commonly used that we need to list them up front. Here is the short list of descriptions, in roughly the order in which you will encounter them.

  Notice that Lightweight Directory Access Protocol includes the word “protocol,” but nothing about an API. LDAP is defined by Internet Engineering Task Force (IETF) standards, called Request for Comments (RFCs). There are several versions of LDAP. Usually, when someone says LDAP today, they mean either LDAP v3 (version 3, which is just a set of RFCs), or they mean the C programming interface for LDAP (as defined in the Informational IETF RFC 1823, for example). In this book, we indicate that we mean the programming interface versus the “wire protocol” by *LDAP C API*, or just *LDAP C*. You can find the actual RFCs in Appendix H and on the CD.

- **Object and Entry**
  Every “thing” in a directory is composed of (at least) a set of attribute-value pairs. So if there is a user “thing” called Graham in the directory, it consists of attribute-value pairs, such as “emailID = graham@somecompany.com” (emailID is the attribute and graham@somecompany.com is the value). Unfortunately, some people call these “things” *objects* and other people call these things *entries*. In some directory services, the things in the directory resemble objects from object-oriented programming (they have methods, for example), and in other directory services the resemblance is a little more remote. In the context of directories, for now, please treat *object* and *entry* as the same.

- **Attribute, attribute type, property**
  As mentioned, every directory entry (or object) is just a collection of attribute-value pairs. As with object and entry, not everyone uses the same terminology for *attribute*. For now, please treat these terms as equivalent. Every attribute (or property) is defined to contain a certain kind of data, referred to as its *syntax*. A user object, for example, might be defined to contain the emailID attribute (so the user Graham can have his emailID in his directory entry). The emailID attribute may be defined to be of syntax Unicode string. It is possible for attributes to be multivalued (you might have multiple e-mail IDs, for example).
- **Schema**

  How do you know that a user object can contain the emailID attribute? Each possible type of entry in the directory and its attributes is defined in the schema.

- **Object class, objectclass, schema class**

  The term used for a schema definition for directory objects (such as when defining what user objects can contain) is usually one of these. Objectclass is particularly descriptive, since it literally means a class (sort of like in OO programming) from which objects can be later instantiated. Somewhere in the directory schema, for example, is an objectclass that defines user objects, so that we can create specific user objects, like Graham.

- **Common name, cn, CN**

  Common name is just an attribute. It is a particularly common attribute (that’s why it isn’t called uncommon name), frequently used in composing names for directory objects. For example, cn=Graham just means that the common name attribute for some entry is Graham (it is probably a user entry in this case).

- **Distinguished name, dn, DN**

  In a file system, every file has a fully qualified path name that uniquely names the file in the context of a particular file system. In a directory, every object has a unique name that uniquely identifies it in the context of that particular directory-naming scheme. The unique name for a directory object is called its distinguished name. An example DN might be, cn=Graham,ou=Sales,dc=Somecompany,dc=com (a salesman named Graham, in an organizational unit called Sales, in the domain Somecompany.com). “ou” and “dc” are other attributes, commonly used in composing DNs, that stand for organizational unit and domain component. There is also a URL format for names, such as LDAP://cn=Graham,ou=Sales,dc=Somecompany,dc=com.

- **Relative distinguished name, rdn, RDN**

  As in file systems, an object has a fully qualified name (such as cn=Graham,ou=Sales,dc=Somecompany,dc=com) and a relative distinguished name. An example of a relative distinguished name is cn=Graham. You can’t really tell from the RDN alone where in the directory hierarchy an object resides, just like you can’t tell where in the file system the file mydata.txt resides.
■ White pages
The white pages in a phone book are where you go to look up a specific entry, such as Graham, to get some specific information, such as his phone number. In computer directory services, performing a “white pages lookup” consists of looking for the attributes of a specific object.

■ Yellow pages
In a phone book, you look things up in the yellow pages by topic, such as plumbers. That is, you look up the general description and get information on lots of plumbers. In a computer directory service “yellow pages lookup,” you perform a search and expect to get back potentially many objects. If Graham happens to be a plumber, the Graham user object might be one of the user objects returned if you perform a yellow pages lookup on “all user objects whose job title attribute is equal to plumber.”
This short introductory section discusses where we are in the evolution of directories and how we got here. It explains why directories exist and what problems they were invented to solve, the origins of directories, and the definition of a directory.
Chapter 1
INTRODUCTION TO THE PROBLEM

“Ubiquitous, complete, and instantly accessible information about every employee.”

Within this simple quotation lie the roots of one form of directory services. Every company that ever grew from a handful of employees to an office full of employees has published a directory. Long ago it might have been a folded paper directory, diligently typed and mimeographed by the faithful receptionist. A really nice one might have been stapled in the middle.

A User-Oriented View

More recently, the company directory was contained in a computer file, diligently typed by the faithful office manager’s assistant. The computer file version was probably copied too (the mimeograph had been traded in), so that everyone could have his or her own. Even more recently, the computer file-based directory was shared or simply sent to everyone, since many office workers had access to a terminal, or perhaps even a computer of their very own. And this is where the real trouble started.

Flat text computer files are the electronic analogue of the typed paper directory. As soon as someone realized that the file was too large to treat like a paper directory, they naturally wanted to build a better user interface than a text editor. Unfortunately, there were problems with treating the directory file as a perfect analogue to a published paper directory. Computer files come in different formats, with different underlying codes (ASCII, EBCDIC, various code pages, and Unicode, for example). Also, not everyone wanted to use a text editor, not everyone should have access to all of the information, and, if there were multiple copies, how would anyone know which was most current? On the other hand, if there were only one copy, it might get lost or corrupted. Directories needed to evolve. (See Figure 1.1.)

People began to think it ought to be possible to have an electronic directory that nearly everyone could use, regardless of the computer they had. They needed a directory that included information about suppliers, products, and sales, as well as
about employees, and even pictures and more advanced media. It would be even nicer if there were powerful search capabilities built into this service. Of course, as this service grew larger and larger, more and more people would depend on it always being available (“Johnson, get me the head of the Venezuelan sales office immediately,” “Ah, I can’t, sir. The computer is down.”).

See Chapter 24 for a case study on using a directory service to solve the classic employee directory problem.

**O ver and O ver**

The basic problem and the basic solution are so, well, basic, that the scenario has been played out many times in many places. Companies, schools, government agencies, and private organizations all stumbled their way through the Directory Dark Ages in basically the same way. That is, a need for information, followed by the obvious solution, followed by the problems of data sharing. Since this scenario played out so often, we now have a plethora of computer directories. This in turn created a second-order problem of coordinating information in multiple directories, and a third-order problem that a directory must be more than a container of passive data: it must be an active manipulator of data.¹ That is, the directory needs to be a directory *service*.

---

¹ Examples of active manipulation (instead of just storing and retrieving) could include managing data relationships, sorting, and possibly even running “agents” or other user-defined functions.
Data Sharing and Manipulation

There are many types of data and data sharing. Companies wanting a list of their employees and telephone numbers create “white pages” (like the white pages of a telephone book), which is just one use of directory services: Other uses of directories also have data sharing at their core. Because directories help solve many data-sharing problems, directories are becoming commonplace.

Achieving a directory that offers “ubiquitous, complete, and instantly accessible information about every employee” involves solving data-sharing problems. We can solve data-sharing problems two ways. The first way is to show how directories have solved problems so that you can apply these same solutions to other problems. The second way is to show you how to let someone else solve the problems for you—by using a directory service.

A System View

“Too much information and too many different ways to manage it.”

Starting off this chapter, we took a user’s-eye view of directory services: an employee directory. Directories are not limited to this “white pages” point of view. A directory is also a place to keep, sort, and otherwise massage information. In fact, if we look hard, we find that many things that an operating system manages fit the profile of the “ubiquitous, complete, and instantly accessible information about every employee” problem.

Location, Location, Location

A directory is a wonderful place to keep system information used to maintain and make the operating system or applications run more efficiently. A good example is locating services. Suppose your company has a critical business application that all salespeople need to access every day. If yours is a large company, this application might be distributed and running on many servers. How does the salesperson’s laptop find a running instance of that critical business application? You don’t want to hard-code it into the application that runs on the salesperson’s laptop. After all, you need flexibility in where the services run, how many servers are running, and even whether employees have to use the same computers. The solution can be to use a directory as a service locator service. (See Figure 1.2.)
A very common and very expensive problem in managing information services is the deployment and subsequent maintenance of applications. Imagine the pain (many readers will know it well) of having to visit every laptop, server, or desktop system to install some spiffy new application. Let’s see, it takes 15 minutes to install it, times the company’s 1,000 machines; add travel time and the time needed to put together help desk problem scripts, and probably a few other things as well. You could try telling people to install it themselves, but in practice that leads to lots of problem calls. It’s enough to make some people quit the business.

Installing and configuring applications can be done from a common administration point, such as, surprise, a directory. If an “agent” can be made to run on every server, or laptop, or desktop, and this agent can peek into a common directory every time the machine boots, then the agent can take care of installing and configuring applications according to the instructions left for it in the directory. Such an agent might even be built into the operating system itself (this is similar in effect to what Windows 2000 does with published applications).²

². An agent such as this could do more than just peek into a directory when a machine boots. For example, it could listen for events generated by a directory service, or by other agents.

Figure A.2 Directory as a service locator service
Why stop with applications? Why not install, configure, and maintain client operating systems from a directory? It is often desirable to lock-down certain OS or other application features so users in certain groups can’t break vital business applications or the operating system itself. Why not even provide the administrative capability to lock-down workstations entirely, so that new applications or OS maintenance can’t be installed unless authorized from above (power users exempted, as usual)? All of these things could significantly reduce costs associated with maintaining systems for many users and all of these things could be done using a directory service. Not every company has a directory service strategy, so you first have to develop one (it isn’t as easy as that makes it sound: see Chapter 29, for example). Chapter 26 is a case study showing how to manage applications using a directory service.

A Unifying Force

A standards-based directory is by nature operating-system-neutral. It is a place to keep information irrespective of what operating system the directory server or the directory-using client is running. This makes a directory a unifying information structure for heterogeneous computing. (See Figure 1.3.)

3. Operating system discussions frequently degenerate into “religious wars.” Rather than going down that road, let’s just say that directories are a good way to bring everyone together into a harmonious computing environment.
Heterogeneous distributed computing isn’t all that hard—just open a socket and away you go. Doing heterogeneous computing with enterprise characteristics, however, is quite hard. There are many vital enterprise characteristics, and one of the most vital is security. A client and server need to know with some certainty each other’s identity and that their communication is secure enough to meet requirements.

A directory service needs to respect key enterprise software characteristics, such as security, or else it is just another “open a socket and away you go” undergraduate term assignment. In particular, the directory needs to understand client and server identity. But what really is identity? I know me. My operating system knows my user ID. What should the (operating-system-independent) directory know? Better yet, how can the directory help me achieve a single identity, instead of one per operating system or application that wants to verify that I am who I say I am?

Identity

In many ways, computer security grew up the same way directories did: Everyone saw the problem and everyone invented his or her own solutions for it. That’s why you need a separate identity for every operating system and for some applications yet more identities on top of even those. Thus you often can’t log on to an Internet site, verify yourself (for example, with a user ID and password) and then go to another Internet site and have your proven identity flow with you automatically—you probably have to enter another user ID and password. That can be annoying.

Identity is, after all, just data about you. Why not have the identity data reside in a directory? Then when each Web site, vital application, or even operating system needs to know who you are, it can go to one common place to get data about you. This concept is called unified identity and directory-based authentication. Not everyone wants one identity, and most companies will not trust such an identity across company boundaries, but in limited doses a unified identity is a good thing and a directory can help to achieve it. (See Chapters 25 and 27 for case studies in Internet personalization and common identity.)

Enterprise Characteristics

Security is a vital enterprise software characteristic. You don’t either have security or not. Rather, security is a spectrum and different applications fall into different spectral lines. The same is true for other vital enterprise characteristics, such as reliability, availability, scalability, serviceability, and performance. An application can’t be said to be scalable or not scalable. Rather, an application has a certain scalability envelope. It
might scale well to four processors, but not to 1,024. Whether that application’s scalability is acceptable or not depends on the requirements.

Meeting requirements for enterprise software characteristics is a key concern for any vital business application or operating system, within the envelope of its requirements. As directories become a focal point (and a unifying force), their support for enterprise characteristics will be a key to their success. Since other key business applications will be built requiring directory services, the enterprise characteristics of those key business applications depend on the directory. If the directory service is not scalable enough to meet the requirements, then the scalability of a vital application that depends on the directory service will be limited. If the directory service isn’t highly available, then applications requiring it can’t be highly available. And so on.

**Bringing the Views Together with LDAP**

From both a user and a system view, directories are very powerful solutions to common data-sharing problems. It’s been widely reported that a typical large company has over 100 directories. They aren’t all general-purpose directory services—many are embedded in products in such a way that the company doesn’t know they are there as directories.

**Lightweight Directory Access Protocol**

The fundamental problems that directories solve are so pervasive it’s no wonder so many directories exist and that some products embed a directory service for their own use. It is a wonder that it took so long to arrive at a reasonable standard for directory services. However, we have one today. It’s called Lightweight Directory Access Protocol (LDAP).

LDAP is two things. First, it is a protocol, or a description of what “goes on the wire.” As a protocol, LDAP describes a syntax for transmitting messages to and receiving messages from a directory service. Second, LDAP is a C language API. Strictly speaking, the protocol is a standard (through the Internet Engineering Task Force) and the C API is an “informational” standard. You could build other APIs in other languages that speak “LDAP on the wire,” and as long as they are implemented according to the standard, the directory server won’t know the difference (for example, the directory server doesn’t know that you are using Java and JNDI versus the LDAP C API). That’s the distinction between the protocol and the API: The directory server sees the protocol and the programmer sees the API.
Since there are so many directories from so many vendors, a key feature of LDAP is interoperation among the directory implementations. Interoperation is the way data is shared not only with users and systems, which a single directory can do, but between directories as well. When different directory implementations can interoperate using LDAP, the sophistication of directory-exploiting applications balloons. For example, if you can get information about all of your suppliers and their products by linking their directories to yours, you have a key piece of infrastructure to help effectively manage your supplier relationships. (See Figure 1.4.)

We discuss LDAP the protocol(s) and LDAP the C API in greater detail in Chapter 21. For now, let's summarize how we arrived at today's directory services by looking at the basic directory requirements.

**Figure A.4  Interoperating directories**

Since there are so many directories from so many vendors, a key feature of LDAP is interoperation among the directory implementations. Interoperation is the way data is shared not only with users and systems, which a single directory can do, but between directories as well. When different directory implementations can interoperate using LDAP, the sophistication of directory-exploiting applications balloons. For example, if you can get information about all of your suppliers and their products by linking their directories to yours, you have a key piece of infrastructure to help effectively manage your supplier relationships. (See Figure 1.4.)

We discuss LDAP the protocol(s) and LDAP the C API in greater detail in Chapter 21. For now, let's summarize how we arrived at today's directory services by looking at the basic directory requirements.

**What, Exactly, Is a Directory?**

A directory is a data storage and retrieval service. It includes APIs that are translated into some protocol to get data into and out of the directory. The directory needs to have some sophisticated features, such as powerful searching and organizing abilities, and it must conform to certain principles for “important” or enterprise software. In the context of the Internet, directory services are a very important underlying service. Most of the rest of this book elaborates on these themes and provides examples of how directories can be used to solve problems.
Where to Find Code

If you are itching to write code, this would be a good time to review Appendix A (for LDAP C) or Appendix B (for Java and JNDI). In addition, Chapter 4 of the Redbook, Understanding LDAP (on your CD), will get you started with the basics of LDAP C and JNDI.

A summary of LDAP APIs is in Chapter 2 of the Redbook, OS/390 Security Server: LDAP Client Application Development Guide and Reference (on your CD). The complete JNDI specification is also contained on your CD (see the home page).

Summary

We started by looking at the problem of data sharing from a user’s perspective and then from a system perspective. Given what users and systems needed to do, we developed a short list of directory requirements (shared data, a format that all can understand, a protocol for getting information into and out of a server, and an API). However, if we started from scratch with just those basic requirements (adding enterprise characteristics, powerful search capabilities, etc.), we could build anything from a database to a transaction processing system to a network operating system. The requirements are extremely broad and deep. So what distinguishes a directory from a database, for example? In the next chapter, we refine our definition of what a directory is and is not, and discuss where you should use a directory versus where you should use a database.
Chapter 2

WHAT DIRECTORIES ARE AND ARE NOT

Not everyone gets the concept of a directory service at first. Sure, everyone can understand the typed department phone list, but not everyone sees the generic problem and the bigger solution that is useful in so many contexts. That is, they can see a directory, and maybe even a directory server, but they fail to see a directory service.

Directory Gestalt

Many operating system designers, application designers, and even distributed computing professionals disagree on what a directory really is. But at some point in studying directories, you will have the “Gestalt Aha” experience. You will be enlightened. You will not only know what a directory is, but you will see solutions to problems—you will see the service part of directories.

This chapter is a short description of what directories really are, and equally important, what they are not. We attempt to describe what directories do, but because new capabilities are being added all the time, the definition of a directory is constantly changing.

Server and Service

Understanding the distinction between a server and a service is vital. A directory server is one process on one machine. A directory service may be composed of any number of servers on any number of machines. It is the service that interests us here; the servers are just pieces that implement it. (See Figure 2.1.)
The Importance of RASSS

Reliability, availability, serviceability, scalability, and security (RASSS) are enterprise software characteristics. Here are brief descriptions.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability</td>
<td>It worked once, it will work again, the same way.</td>
</tr>
<tr>
<td>Availability</td>
<td>It is always there to respond to my requests.</td>
</tr>
<tr>
<td>Serviceability</td>
<td>When something goes wrong, I can find out why and fix it.</td>
</tr>
<tr>
<td>Scalability</td>
<td>When I add another processor, performance improves.</td>
</tr>
<tr>
<td>Security</td>
<td>I can limit or allow access as required and generate audit trails.</td>
</tr>
</tbody>
</table>

A program exhibiting a high degree of RASSS characteristics is more suitable for mission-critical business applications than a low RASSS program. That is, high RASSS equals high robustness.

A directory service must exhibit certain RASSS characteristics, but perhaps to a different degree than a real-time operating system. Availability, for example, is achievable several ways. Where an operating system often maintains availability natively (in one instance of itself), a directory service often maintains availability by maintaining many copies of the same directory data in different servers on different machines. So availability can be achieved either with highly robust code, or with redundancy, or both.

A directory service has to exhibit a degree of scalability that exceeds that of many operating systems. This is because the directory service is available to many operating system instances simultaneously. If the directory is critical to business applica-
tions across several systems, it is reasonable to think that the directory must be as scalable as the sum of its users' scalability.

So RASSS is vital to directory services, but in subtly different ways than in an operating system, or perhaps even a database.

**What Directories Do**

The easiest way to define a directory may be to look at what directories do today. This is a post hoc argument, but we are after a working definition, not a mathematical proof.

A directory keeps “read mostly” data. Your home address is a good example: It changes, but not very often and certainly not every few seconds. Another good example is your computer identity as known by an Internet bookseller. Again it could change, but not often. So how often does data have to change before it is considered inappropriate for a directory?

In the directory world, we classify data based on their rate of change compared with the ability of the directory service to absorb the change. That is, how often data changes in a directory is compared to the replication frequency of the directory service (for a definition of replication, see Chapter 19, Replication and Partitioning). If data changes at a rate of once every second, but one directory server updates its satellite server partners only every ten minutes, there is an impedance mismatch between the data and the directory service. (See Figure 2.2.)

---

**Figure B.2** Impedance mismatch between the user's desire (update every second) and the directory service's configuration (update every minute)
Functional Characteristics of a Directory Service

One characteristic of a directory service is that it must support a level of RASSS that meets its requirements. Here is a brief list of the functional characteristics of a directory service.

- Holds any type of object, although some are not generally suitable (see What a Directory Is Not, later in this chapter)
- Can add new types of objects (see Chapter 16, Schema Fundamentals) in a structured way
- Makes its contents available to users across many different operating systems or applications, possibly in geographically dispersed places, and probably via the Internet as well as corporate intranets
- Locates and searches its contents in a powerful way

It’s a pretty simple list until you add the RASSS requirements and the other general requirements of any enterprise software (that it be administerable, for example); then it becomes a tough nut to crack.

What a Directory Is Not

To enhance the definition of a directory, let’s examine some things that a directory specifically is not.

Remember ACID?

No, this is not a discussion about the 60s. ACID is the database acronym for atomicity, consistency, isolation, and durability. Maintaining ACID properties is the driving force for database and transaction people. The properties are defined as follows:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomicity</td>
<td>All changes either happen, or don’t, with no partial updates.</td>
</tr>
<tr>
<td>Consistency</td>
<td>The state of the data is always “correct” and predictable.</td>
</tr>
<tr>
<td>Isolation</td>
<td>Data changes appear isolated from other concurrent changes.</td>
</tr>
<tr>
<td>Durability</td>
<td>Once a change is made, it will persist (it won’t be “lost”).</td>
</tr>
</tbody>
</table>

Where do these things fit in the definition of a directory? For example, isolation means that if you change data, it appears from your program’s perspective that other changes happened either before yours or after yours, but they don’t affect you in mid-transaction. We defined a directory to be a very large, very available repository
of information on just about anything. Isolation in this environment can be hard to implement and will consume a certain amount of computing resources.

For example, if you own a database and it sits in a file and you control all access to it through a software process, you can guarantee that when a program accesses data, it appears that the data do not change “out from under it.” This isolation is achieved with locking and you can do it without too much trouble when you control all access to the data (well, actually, it is a lot of trouble, but it is done by every database).

Now imagine that the data is in a distributed directory. The directory not only has multiple replicas, but it might even be replicated across different vendors’ directory servers in different countries.

See the problem? It’s about scope and what you are willing to pay. What is your required scope of ACID properties that solves a particular problem, but that doesn’t cost too much to implement, or cost too much in runtime resources?¹

**Not Quite Nirvana**

Sure, you would like your directory to have all of the characteristics of a database and a transaction processing system, but it won’t. Or at least it won’t until you are willing and able to pay the overhead associated with maintaining the ACID properties with acceptable performance or the price tag on various technologies finally reaches a point where it is cost-effective to keep all data in a database. Surely there must be something that isn’t quite ACID but that solves your data-sharing problems. There is: a “loosely consistent” directory service.

**Loose Consistency**

Consistency (the ‘C’ in ACID) is quite expensive. Is consistency really required for a “read-mostly” data store? Well, yes, it is, but perhaps not to the utmost extent.

Suppose the data that you are changing is your home address, stored in a directory. Is it vital that your home address be 100 percent accurate in all 99 instances of the directory server that make up the directory service spread throughout the corporation at exactly the same atomic moment? Probably not. No one will die during the 15 minutes it takes for your address to propagate throughout the enterprise-distributed server farm. Your address will be consistent, eventually. This is called loose consistency.

---

¹ For a brief exhibition on why distributed ACID costs so much, see “Two Phase Commit and Recovery Logging,” in Chapter 5 on the CD.
Maybe a better term would be *temporal consistency*, but loose seems to be in the common directory vernacular.

If you want to keep your address 100 percent consistent, you could keep it in a central database and anyone wanting to modify it would have to make a database connection and perform a transaction on it. That works and lots of data fit that kind of model, but it can be expensive. Databases consume resources when ensuring ACID properties, and the expense can rise proportionally with the amount of data managed and the number (and type) of clients that need to access it. And ACID isn’t the whole story—there are also performance and the other enterprise characteristics to worry about.

When databases *can* relax their consistency rules for certain kinds of data, they begin to look like candidates for directory services. In fact, some directories are built on top of relational databases. For example, there’s an LDAP directory server built on top of a relational database on your CD: the IBM SecureWay LDAP Directory Server.

**A Definition Can’t Be Wrong— by Definition**

Any definition of directory will almost certainly be wrong over time (it will have deteriorating correctness). It will be wrong in the long run because, as technology evolves and gets even cheaper, it will be possible and desirable to have more of the ACID properties and other features in the directory service.

**Good, Bad, and Ugly Data**

We have had a lot of discussion about what a directory is and isn’t. Here, we attempt to clarify the role of a directory service based on the kinds of data that belong in it. Some kinds of data are a good match for the “not quite ACID” and “pretty much RASSS” directory technique.

<table>
<thead>
<tr>
<th>Good Match</th>
<th>Why</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Low change rate, low ACID requirements</td>
</tr>
<tr>
<td>EmailID</td>
<td>Low change rate, not vital</td>
</tr>
<tr>
<td>Name to IP map</td>
<td>Low ACID requirements (this is DNS)</td>
</tr>
<tr>
<td>Location of X</td>
<td>If X isn’t there, they will try something else.</td>
</tr>
</tbody>
</table>

In the last example (location of X), X can be nearly anything that needs to be found. Imagine a server, somewhere in the network, that returns the correct time. You could simply hard-code the location of this server, but that would make your program “brittle.” A better technique would be to ask a directory service where the time service is and then contact the time service based on what the directory says. This way,
the time service could move and your program would still work fine (an example of improving the enterprise characteristics of your program using a directory).

You might see this “use a directory service to find other servers” argument as circular, unsatisfying, and loose. The clue is “server” versus “service.” A service is always available, and this key characteristic makes a directory service a fine way to locate things, such as servers, services, devices, or even people.

The directory might be able to provide a lot of “value-add” on top of the location services just described. The directory might return a list of time servers, in case one isn’t working. The directory could give you hints about which time server is closer to you and hence able to provide a higher quality time value (due to network delays). It might even be able to tell you which time server is least busy and therefore most likely to get to your request quickly. Taking this example to its extreme, a time service could be built with the assistance of a directory service.

Following are some examples of data that are poor candidates for inclusion in a directory. At the very least you should include them only after seriously considering the application requirements and the capabilities of the directory service.

**Bad Match**

<table>
<thead>
<tr>
<th>Why</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank account balance</td>
</tr>
<tr>
<td>Heart rate</td>
</tr>
<tr>
<td>Missile target reference</td>
</tr>
</tbody>
</table>

The last example (Missile target reference) doesn’t change often, but when it does anyone using the data must have it immediately. This is referred to as *change latency*. If the missile target reference changes and someone needs to use that data now, he or she can’t wait for the data to propagate to other directory servers. In the heart rate example the data changes so often that it overruns the time required for the individual directory servers to coordinate and make their copies consistent (an impedance mismatch). Finally, the bank account example requires ACID properties; that is, if someone cleans out his or her bank account, it shouldn’t be possible for an accomplice to clean out the same bank account at another branch (taking advantage of the loose consistency of the directory data).

Following are examples of things that most definitely don’t belong in a directory. These would be ugly situations.

**Ugly Match**

<table>
<thead>
<tr>
<th>Why</th>
</tr>
</thead>
<tbody>
<tr>
<td>Federal Reserve transactions</td>
</tr>
<tr>
<td>Shuttle navigational data</td>
</tr>
</tbody>
</table>
Why don’t these things belong in a directory? The Federal Reserve Bank of the United States has rather strict requirements on transfers between itself, the 12 regional banks, and the U.S. Treasury. It ruins their day to misplace funds. You know the old saying, “a billion here, a billion there, and pretty soon we’re talking serious money.” These kinds of transactions are much more serious than the very limited type that you are likely (or allowed) to perform on your checking account, and consequently there are very strict controls and accountability. The example of navigational data for the space shuttle certainly does not belong in a directory. Most directories are simply not “man-rated.”

Man-rated software is that on which someone’s life may depend. Like the Federal Reserve Bank example, this is a very special kind of processing and different controls are required (see Chapter 5, Man-Rated, Enterprise-Rated, Good-Enough, and Don’t-Care).

A Brief Code Example

When all you have is a hammer, every problem looks like a nail. When you are a software engineer, you write code. Let’s look quickly at an example program that uses a directory.

Just how do you get data out of a directory? There are many possible APIs to use. Since LDAP is an IETF standard, we use it extensively throughout this book (via LDAP C, or JNDI, or other APIs). Here is an invocation of an LDAP C API called ldap_search_s(), which is used to search a directory synchronously (that is, we wait for the return).\(^2\)

\[
\text{rc = ldap_search_s(ld, } \text{ _T("dc=dehco,dc=com"),} \\
\text{ LDAP_SCOPE_SUBTREE, } \text{ _T("(objectclass=organizationalUnit)")},} \\
\text{ NULL, } \text{ FALSE,} \\
\text{ &result);} \\
\]

2. When the term *man-rated* was invented, it referred to American astronauts, who were then all men. The term is still used, but of course refers to anyone.

3. If you are not familiar with the LDAP C API, please refer to Appendix A for a brief introduction. Appendix A includes an API summary that will give you the feel for it. If you desire a more complete LDAP API treatment, see the Redbooks on your companion CD. While more difficult to understand, you can also see IETF RFC 1823, an informational RFC for the LDAP C API (also on your CD under Appendix H).
This is looking for any object in a directory located on the server dehco.com, or attached to dehco.com in a tree structure that is derived from a specific schema class (objectclass=organizationalUnit). This one API will return (in the result parameter) a list of all organizational unit (OU) objects on that server or below it in the tree of servers (if a tree exists, which you can’t tell from this code). If you don’t know what an OU is, don’t worry, it’s just a way to organize people or other administrative entities. Think of an OU as a container of things to manage alike.

You can find the complete program, called “simple,” on the CD. If it is your first foray into LDAP, you might not think it is so simple. As with many examples in this book, the CD contains both the code and screen captures showing the program working.

First, this example shows some common LDAP C APIs. Second, it shows how to create a new entry (object) in a directory. Third, it shows how to find specific entries in a directory. Fourth, it shows how to dissect the returned results of a directory search. In short, this example encapsulates the primitives involved in creating and searching for directory data.4

One thing this example does not do is define new types of entries for the directory (for example, extending the schema for a new objectclass instead of using the pre-existing OU objectclass). Have patience. We modify the schema in later examples.

While we now have a working description of a directory service, and even an example of how to access a directory’s data, we can’t pretend that it is a real definition. In the business of software, things change a lot and they change fast. Even if things didn’t change, a solid definition of directory would be difficult to write down. After all, a directory is, in a way, a poor man’s database and transaction processing system, built as a stepping stone to the day when we can afford ACID RASSS everywhere, throughout the Internet and all corporate intranets.

For more on what directories might be capable of in the future, see Chapter 14, As Directories Grow Up.

**Summary**

In this chapter, we attempted to define what directories are and are not. A directory is not quite a database, not quite a transaction processing system, and not quite a file system, although it shares certain characteristics with all of these other types of infor-
information processing. In the perfect world, a directory would provide the letter and spirit of the ACID law (atomicity, consistency, isolation, and durability) and conform to the highest standards in all the enterprise software characteristics (RASSS—reliability, availability, scalability, serviceability, and security). But the world isn’t perfect.
Chapter 3
DIRECTORIES ARE EVERYWHERE

In today’s computing environments, directories can be found everywhere. There are directories for users, files, and configurations. In this chapter, these directories are described in order to give you a better understanding of the issues to be addressed using a directory service.

Directory Forms

It seems that for as many operating systems and applications that are created there are just as many or more directories created to support them. This has produced an environment in which directories can be found everywhere one looks. They range from simple configuration files for applications to generic, network-enabled directories that are shared by many users and applications. The key point of this chapter is to raise your awareness on the types, intent, and uses of different directories in your environment today.

System Directories

As briefly described in Chapter 2, every multiuser operating system contains a mechanism—a system directory—for defining the users (and sometimes groups of users) that can access and interact with the operating system. A system directory is focused on the security aspects of defining an identification to be used on the operating system. A system directory typically defines a primary key for users and groups, and stores a set of information about each that is specific to the security model used by the operating system.

A system directory might also encompass the set of information required to initialize the operating system and its services. This portion of a system directory is usually made up of configuration files, used to store printer configurations, file system configurations, and other peripheral settings. Input/output configurations, device settings, printer names, print queue names, file system mount points, and catalogs of
disk and volume information all become part of a system directory. As Figure 3.1 shows, the complex set of information that makes up the system directory is hard to decipher and difficult to work with.

One of the biggest problems with system directories today is that more and more features have been added, and more and more mechanisms have been defined for storing this information. As a result, configuration information is scattered around the system, managed by multiple interfaces, has subtle interdependencies, and generally must be administered by experts. To compound matters, the format and features of system directories are highly operating system dependent, so the skills required to administer one operating system are completely different from the skills required to administer another. Surely in this era of computing, configuration and administration of the operating system should be a small side task, not a full-time career.

**Phone Book Directories**

As the size of organizations has increased and electronic communications have become more widely used, maintaining printed versions of information has become impossible. The sheer volume of change between when something is printed and when it is actually used renders the printed form of a phone book, for example, frustrating to use and even harder to maintain.
Phone books represent another important type of directory: a directory that is used as a lookup service. Phone books contain information that is used to contact someone or something else. The form of phone book that we work with every day is still a printed version—the phone book that lists people and businesses in our local areas. Typically maintained by the local phone company, this phone book is produced from the phone company’s customer records. If you don’t have a phone, you don’t show up in the book. But the book is incomplete and frequently out-of-date—we’ve all come to live with the possibility that the person or business’s phone number or address might be unlisted or have changed since the book was published.

Recently, a number of companies, affiliated with some of the larger communications companies, have begun publishing the information found in the phone book on the Internet. (See Figure 3.2.) These lookup services are typically not much better than the printed phone book, since the information is not always updated as changes occur. Still, an on-line phone book lets us locate a hardware store in Atlanta from the comfort of our own home in Chicago.

Phone book information is generally organized into white pages and yellow pages. (Refer to Chapter 1 for a description of white pages and yellow pages types of information.) Whether we use the white pages or yellow pages depends on the type of search that we would like to perform against the information. Unlike the printed form of a phone book, a white pages lookup in an electronic directory is merely a different type of search than a yellow pages lookup. In a hard-copy phone book, information is repeated on different pages of the book.

A variation of the phone book style directory is the library card catalog. A library card catalog provides information on how to locate books, periodicals, and reference information in a particular library. The Library of Congress Classification system is used to uniquely identify books in the library, and library shelves are ordered by the Library of Congress Classification system. The Library of Congress Classification system identi-
fiers are hard for humans to remember, however, so a card catalog was created. The card catalog (and the electronic form of the card catalog) offer white pages and yellow pages forms of lookup for books in the library. The white pages lookup of a book is the portion of the card catalog that provides a search by title. The yellow pages lookup of a book offers two types of searches: by subject and by author. So while the yellow pages lookup in the phone book is by service offered, yellow pages lookups in the library can be performed by subject or by author. Like the phone book, hard-copy forms of the library card catalog are nightmares to maintain, since adding a book requires adding multiple cards to the card catalog. Removal of a book from circulation involves finding all the subject-based cards.

As companies have moved their corporate phone books into electronic form, maintenance of them has become much easier. Changes to an employee's information can be controlled by the employee, the human resources department, the mail system administrator, and the physical services administrator. But the corporate phone book is only one of the places where information about people and services is located. And even though the information might all be stored in electronic format, synchronization of this information with the multitude of other directories in the organization is a problem.

Another form of the phone book style of a directory is the directory that is used to locate and contact other computers in a network of connected computers. Directories of this form are typically tied to the particular network implementation, because naming and addressing computers over networks is highly dependent on the network implementation chosen. These directories handle name-based or white pages style lookups for computers in the network. Some of these directories also offer service-based lookups, which is more like a yellow pages style lookup.

**Application Directories**

Just as operating systems have come to define information about the users, groups, and peripherals that are attached to the system, the same types of information are contained in directories that are specific to applications. It seems that every application has invented its own form of directory. These are sometimes very different from one another, even between the same applications that run on different operating platforms.

An application directory usually contains the information about users, groups, and peripherals that is specific to the application itself. The application might store the last printer selected and the settings for it or the preferences that a particular user has set up for his or her interaction with that application.

Sophisticated applications might be able to contact peers in the network or other applications in order to exchange information. The details of these connections are also
stored in the application directory, in a proprietary format that is convenient for the application but not so convenient for administering the information. (See Figure 3.3.)

With each application directory comes a different file format and different user interface. Access to this information might be limited to the machine on which the application is running. Some applications even go so far as to always look in fixed spots on a particular system, thus constraining the number of instances of the application that can run on the machine at any one time.

**User-Oriented Directories**

Many characteristics of directories are based on the intended users of these particular directories. Unfortunately, creating different implementations for each form makes the maintenance of this information difficult, if not impossible. Some uses of directories are introduced here. The users of directories are covered in more detail in Chapter 7.

**Personal and Mail Systems**

A personal directory comes in a number of forms. Small organizers can hold hundreds of names, addresses, e-mail addresses, and phone numbers that are important to an individual. Mail systems typically offer a personal address book for storing similar
information specific to the particular mail system. (See Figure 3.4.) A small industry has sprung up around producing sync agents between personal devices and personal computers so that information important enough for a person to carry can also be entered and maintained in one location. Even with these advances, administration of one’s personal address book(s) is still a time-consuming and tedious task.

Mail systems make use of a phone book style directory to manage mail addresses (physical and electronic) and phone numbers. In addition, mail systems typically offer a grouping mechanism whereby mail can be sent to all members of a group simultaneously. (See Figure 3.5.) In addition to mailing addresses and group information (commonly referred to as distribution lists), mail systems also need to store information about how to send, receive, or forward information coming to them.

Depending on the mail system being used, some mail systems allow groups to contain other groups. Fundamentally, a distribution list merely lists names. Specifying that a mail message be sent to a distribution list causes all names in the distribution list to be sent the message. If distribution lists can contain other distribution lists in the set of names, then these distribution lists are then expanded as well. Distribution lists allow frequently addressed sets of people to be sent mail as a unit. Allowing distribution lists within distribution lists means previously defined lists can be re-used to build larger lists.

![Figure C.4 Mail clients and personal assistants have directories](image-url)
Frequently the information stored about individuals and services in a mail system and the information stored in a person’s personal directory overlap. The synchronization of this information is usually the responsibility of the individual. Whether or not this is a problem usually depends on individual records in the personal directory.

**Operating Systems**

Directories defined to allow operating systems to provide services to their users span multiple subjects, file formats, and access methods. An operating system must keep track of storage devices, I/O devices, printers, and other peripherals; their configuration; and definitions of users and groups that can access the system. In addition, an operating system must keep track of individual permissions and access rights to the devices, peripherals, and files managed by the operating system.

Much of the information stored about users and groups is operating-system specific: home directory, default command environment, unique identifier on the system, credential (usually an encrypted form of a password), number of attempted logins, last successful login, last failed login, and more. In many cases, however, some of the information used to define users and groups to an operating system is the same (or only slightly different) from the information used to define users and groups to a mail system or stored in a personal directory. As the number of systems increases in an organization, the number of user identities a person must remember increases, as
does the number of user IDs an administrator (or more likely a set of administrators) must attempt to keep accurate and up to date.

Several products have now emerged on the market that attempt to ease this situation. Meta-directory services, which attempt to synchronize information to and from multiple, separately administered sources are one example. Another form of this unified view of administration are applications that can be used to define a user in a corporation. These applications then push that information out to the individual operating systems on which the user is to be defined. A similar method is used for managing group definitions. (See Figure 3.6.) More details about meta-directories can be found in Chapter 20 and the case study on meta-directories in Chapter 29.

For users of these systems the situation is somewhat more primitive. It is not uncommon for users in an organization to have upwards of ten user IDs for different systems and applications they access as part of their job. This usually means ten passwords as well. To compound matters, each of these systems may be defined to require different password formats, different password change intervals, and different expiration dates. Users either write down their passwords and store them (if they’re conscientious) locked in their desks, or, more commonly, tack them to the corkboard over their desks. Really sophisticated users will store their passwords in their personal directory. This is really not much better than putting them on the corkboard, however, since personal directories are rarely password protected (that would be yet another password to remember!).

![Figure C.6](image_url)  
**Figure C.6** Synchronization using a meta-directory
Networks

For every network protocol that is developed, at least one directory has been created to support it. *System network architecture* (SNA) contains a directory in the form of system names coupled with logical unit names. The combination allows conversations between connecting applications or devices and thus constitutes a naming model—and hence a directory.

In TCP/IP networking, names are mapped to 4-byte addresses called IP addresses. A separate service, accessible using TCP/IP, stores the mappings between names and IP addresses. This name service is called the *domain name service* (DNS). DNS was first introduced to provide a simple name to address mapping so that communications could be established between two applications. DNS has since been enhanced to store all kinds of addressing information. A record in DNS is used by SMTP-based mail systems for sending mail messages to individuals where the mail address contains a DNS domain name. Records in DNS are used to define services that are available in a domain. These service records allow generic domain names to be used instead of a fixed system name, so the service can be relocated by administrators as needed. Information stored in DNS is accessed using a resolver that exists on the local system. The resolver communicates with a DNS server to access information. (See Figure 3.7.)

![Figure 3.7](image)

*Figure C.7*  Host name address lookup using the resolver
Other networking protocols contain similar directories, all used to look up and contact systems, applications, devices, and individuals using the associated network. Because of the differences in address formats and name formats, these directories are built to be specific to the network protocols they are used with.

**Generic Directories**

In recent years many people have contemplated the multitude of directory types and concluded that we need a generic directory, usable across applications, systems, and networks, in order to organize, administer, and make the computing environment easier to use. The similarities and overlap of information across directories seem to imply that using a generic directory could help users, computers, and administrators alike. Furthermore, centralizing this information would eliminate the need for multiple copies, simplify the computing environment, and reduce errors. Figure 3.8 shows several applications using a generic directory service to hold their configurations with the administration of the applications done by updating information in the directory.

In the late 1980s international standards organizations, at the urging of communications companies, began defining generic directory service. At the same time, a new, generic, network protocol was being defined as well. The result of this work was the X Series of standards produced by the Consultative Committee on International Telephone and Telegraphy (CCITT). The X.500 family of standards defines a

---

**Figure C.8** A directory service can support multiple applications simultaneously
generic directory service for use over the open systems interconnection (OSI) networking protocol. While OSI has seen success in parts of the world, it has not become the accepted networking protocol worldwide. Implementing a fully X.500 conformant directory service has proven difficult. Instead, many generic directory service implementations have been created based on the X.500 directory model but that do not implement all aspects of the X.500 model.

A generic directory service is extensible to allow definition of information to be contained in the directory. A generic directory service allows both name-based and attribute-based information searches. In addition, a generic directory service is accessible using a protocol that allows add, modify, delete, and search operations to be performed against information in the directory. Examples of generic directory services available today include X.500 implementations, Novell Directory Services (NDS), Netscape Directory Services, IBM SecureWay Directory, and others.

The value of using a generic directory service to store configuration, user and group, or locating information for an application is realized when multiple applications exploit the same generic directory service. Said a different way, the value of using a generic directory service increases as the number of exploiters increases.

**Corporate Directories**

A standalone corporate directory, one that is not integrated with other tools such as the company’s electronic mail system, becomes frustrating to use. Users and applications wind up cutting and pasting information from one application to another. When, on the other hand, the corporate directory and the company’s electronic mail system are tied together by exploiting the same directory, both are improved. Searches of the corporate directory can be displayed to users in such a way that sending mail to some person or group becomes a point-and-click operation instead of cut-and-paste. An application can use information directly from the corporate directory in formulating electronic mail messages to be sent to people and groups. The case study in Chapter 24 describes such a corporate employee directory. The case study on metadirectories in Chapter 29 provides more information about corporate directories.

**Human Resources Information**

In many companies today, information about employees is scattered throughout a multitude of directories. When instead, a generic directory service is used, all of the pieces of information about an employee can be tied together. Administration can be simplified by organizing the information about an employee, including the human resources (HR) information a company holds. Defining, modifying, and removing
information that pertains to the person can be done across the company in a consistent fashion, using the same interfaces.

**Future Demands**

In order to meet current and future demands for administration and navigation to information and services, applications must begin to exploit a generic directory service. This would ease administration and use for all applications that exploit the generic directory. Users and applications could count on a directory for locating other users and services. Additional demands on applications would also drive their use of a generic directory service.

**Distributed Administration**

Applications are being required to allow greater and greater flexibility in their management. Gone are the days when we needed to be physically near the machines we administered. Indeed, we should be able to do much administration in the future from the comfort of home. This will allow us better response time for fixing administrative problems in application configurations. One way to solve distributed administration requirements is the use of a generic directory service to store application configurations.

**Policy-Based Administration**

Many applications are now being created that, in order to operate efficiently, require the administration and configuration of a number of different machines or programs, spread across many systems in a network. Administration of individual details in such applications is tedious. In response, a new form of administration is emerging. Instead of administering on a resource-by-resource basis, we can establish a set of policies, which are then applied against a large number of resources. Administration of the policy affects the configuration of a potentially large number of resources. To perform this style of administration, we must make policies accessible across the organization, which implies the use of a generic directory service accessible from everywhere in the organization.

**Consistent User Interface**

Administrative interfaces are beginning to converge on the same style of user interface. A navigation pane coupled with a subject pane has become a popular paradigm for administering applications. Tying all information about an organization together
in the navigation pane allows administrators and users to understand the organization and structure of a company’s computing resources. Such an organization of information, which allows the definition of a consistent user interface, can only be obtained when a generic directory service is exploited. The generic directory service is used to organize the computing resources of an organization and to provide a consistent interface on which a consistent user interface across applications can be established.

**Skills Portability**

With a consistent user interface for locating and administering multiple applications, the user interface barrier between using and administering multiple applications is lowered. Skills in interacting with the administration functions of applications can be re-used as new applications are defined, allowing greater administrative productivity.

**Summary**

Directories are everywhere. As applications begin to leverage generic directory service, the value of the directory increases. When computing resources can be located through a generic directory service, end users and administrators can better understand the computing environment of the company and become more productive. The cost of administration and use of the computing resources in a company can be reduced by using a generic directory service. Using a directory opens up interesting possibilities for even greater administrative efficiencies such as policy-based administration.
Early Standardization

Using a directory service is not a new concept. The use of directories for managing location information is well established in many networking protocols. This can be seen in many of the directory services that have been defined in the past twenty years.

SNA Host Names and Logical Units

In the SNA, communication between two endpoints is accomplished using conversations. Conversations are established between logical units running on different machines within the network. Different logical units communicated with different protocols over SNA networks. Logical units (LUs) were defined for simple terminals as well as program-to-program communications.

In order to contact another LU in the SNA network, names are required. Thus, the coupling of host name and LU name constituted an address that could be contacted in an SNA network. While the complete set of names is distributed throughout the network, the presence of these names implied a directory name space of hostname:luname for establishing connections through the SNA network.

Domain Name Service

TCP/IP networks define addresses for sockets. Sockets represent connections in terms of a numeric address, which represents a network interface, and port number, which is unique at the network interface. The notion of a distributed name service was introduced in TCP/IP. It was the job of the name server to take as input a textual name and return a TCP/IP address that could be contacted over a TCP/IP network.
DNS is defined as a distributed naming service. Internet Engineering Task Force (IETF) Request for Comments (RFC) 1035 (see Appendix H) defines the communications protocol used with a DNS server. The DNS name space consists of a set of alphanumeric strings separated by periods. The names increase in specificity moving from right to left (for example, sys1.widgets.com is an address within widgets.com and thus is more specific than widgets.com). The TCP/IP protocol defined for accessing DNS includes a referral mechanism so that if one DNS server does not contain information about the host name that was requested, the accessing application is referred to a different server that might help narrow the search, until, finally, the information is located. This referral mechanism allows multiple DNS servers to be used to cover the entire DNS name space.

Figure 4.1 illustrates the referral mechanism in DNS. A client makes a request to find the IP address for host “www.company2.com” (step 1). Because the local resolver does not have this information, a DNS server is contacted (step 2). The first DNS server responds to the resolver with a referral to contact the server on 3.14.2.2 (step 3). In step 4, the resolver redrives the IP address lookup to 3.14.2.2 and is redirected to a DNS server running on 3.22.2.3 (step 5). In step 6, the resolver redrives the IP address lookup for “www.company2.com” again to the DNS server on 3.22.2.3, which responds with the IP address of 3.22.6.8 (step 7). Finally, in step 8, the requested IP address is returned to the original requester. DNS servers and resolvers make heavy use of caching techniques to speed IP address lookups.

When DNS was defined, the top layer of the name space was fixed. This allowed the name space to cover the entire world and for regional naming authorities to be established in order to avoid naming collisions. As names were allocated, more specific names (increasing the number of name elements on the left) can be defined independently of other names allocated by the regional naming authorities. A number of top-level DNS names have been defined and are administered by the Internet Address Naming Authority (IANA). These top-level DNS names include .com, .gov, .org, .edu, and .net.

As the value of this distributed name service increased, multiple applications began to exploit its presence. MX records were added to the name server to allow SMTP mailing systems to locate a central mail server that handled a subnetwork of systems. Other records were introduced for storing locating information for other services. As the use of DNS for these things increased, extensibility limits of DNS were stressed. A generic TXT record became overused, and it was soon apparent that use of DNS as a generic directory service was problematic.

With administration of DNS information being done primarily by updating DNS configuration files, administration of DNS servers and its distributed name space became a complex job. This enabled high performance retrieval of IP addresses in response to queries, but severely limited the use of DNS as a generic directory service.
Today’s use of DNS is enormous. DNS is used every second of every day as information is accessed over the Internet. The largest use of DNS is centered on its high-performance mechanisms for host name-to-IP address lookup services. Many fully qualified URLs, used to connect the World Wide Web (WWW), contain a host name that must be translated to an IP address.

Dynamic update of DNS information has only recently been defined. Dynamic DNS servers, coupled with SRV records, are becoming useful for finding essential network services. (Service records, known as SRV records, are additional data elements stored in DNS servers and are used to store the host name and port number of services that are available over TCP/IP networks). Notably, generic directory servers as well as
security servers can be located in the network through the use of SRV record. This means directory clients and security clients can use an already established mechanism to locate an initial directory and security server.

**Distributed Computing Environment Cell**

**Directory Services**

As the use of TCP/IP increased, many attempts were made to simplify the definition of networking protocols for creating distributed applications. One such attempt was the distributed computing environment (DCE). DCE was created in an effort to provide a complete, distributed platform on which applications could be written. All applications used a consistent security model (identification and authentication based on Kerberos), a consistent networking protocol using remote procedure call (RPC), and a consistent directory for finding resources using cell directory services (CDS). In addition, a consistent set of distributed time services was introduced to manage the complexity of synchronizing time across a network of machines.

Figure 4.2 depicts a DCE cell environment with three DCE CDS servers deployed. DCE CDS clients, called *clerks* and *advertisers*, dynamically discover DCE CDS

![Diagram of DCE CDS servers](image)

**Figure D.2** Multiple CDS servers manage the cell directory
servers using broadcast protocols. DCE CDS servers replicate information between each other while DCE CDS clerks retrieve information on behalf of lookup requests. In step 1 a DCE CDS lookup request is made to retrieve the DCE RPC binding handle for a “vectorProcessor” service. In step 2, the DCE CDS clerk contacts the most appropriate DCE CDS server, which responds with the DCE RPC binding handle (step 3). The CDS clerk then returns the binding handle to the application for use in connecting to the “vectorProcessor” service (step 4). Caching is used in the DCE CDS clerk to remember DCE CDS server locations as well as to respond quickly to repeated client requests.

The primary use of DCE CDS is in storing and retrieving binding information, which allows RPC client applications to locate and connect to RPC server applications. These lookup operations could be administrator-controlled, allowing the management of applications to be performed independently of the application’s users. DCE CDS also allowed any set of name-value pair style information to be stored. DCE CDS was extensible in this regard. Suddenly the directory could be used to store more information than just host name to IP address lookups. The notion of storing configuration information in a directory service became a reality.

DCE has enjoyed a growing success in the computing industry, especially with some of the applications that have been written to exploit its services. Distributed file service (DFS) and Encina-based transaction processing have given DCE a presence in the industry. Use of DCE CDS as a generic directory, however, has only seen limited success. While its replication model is very granular, the management aspects of CDS are complicated and time consuming.

LAN Managers

While TCP/IP networks between Unix-based machines were being created quickly all over the world, small networks of personal computers (PCs) were being created in office environments, and people began to share the information on their PCs. Shuttling disks between computers and switching printer cables became increasingly inefficient.

A solution to these problems was established with the LAN manager environment. In such an environment, a set of PCs are set up as server machines, providing a centralized user directory, file serving, and print serving. As features were added to these servers, requests between client machines were routed so that printers and disks attached to client machines could also be shared. (See Figure 4.3.)

The LAN manager environment began as an effort to provide file and print sharing among a set of PCs operating in a local network. Today, these networks offer security, directory, file, print, and application services, not only locally but also across the enterprise.
During the 1980s, communications companies began to look toward wide area networking to handle the increasing demand for global communication. In order to address this growing need, a new set of networking protocols was designed called open systems interconnection (OSI). OSI presented a seven-layer model for communications. As part of the set of standards developed by the CCITT in support of the OSI model, a definition of a generic directory service was developed. The X.500 family of standards defines a generic directory service. Using Abstract Syntax Notation 1 (ASN.1) as the declarative language, protocols for interaction with the directory service were defined. Directory service behavior for client access (embodied in a protocol called directory access protocol, DAP), server-to-server communications, replication, chaining of requests between servers, and management protocols for the directory service were all defined as part of the X.500 series of standards. The X.500 series of standards defines the behavior of entities (boxes in Figure 4.4) and protocols between (lines in Figure 4.4) which together make up an X.500 directory service.

The X.500 standards are impressive—so impressive that implementing them has proven to be very complicated. The extensibility aspects of the X.500 directory
model make it an ideal base on which to build a generic directory service. The distributed protocol access and update operations that are defined allow for rich, attribute-based search as well as add, modify, update, and compare operations to be performed against the directory service. The X.509 standard indicates methods by which security can be applied to information stored in an X.500 directory service and the protocol offers a “bind” operation that lets a client identify itself to a server for access control–checking purposes.

A base information model for representing organizations, organizational units, persons, and groups is part of the X.500 set of standards. The X.520 standard defines this base set of X.500 directory schema, which focuses on the aspects of organizations and people that are necessary for locating them in the physical world, such as first name, surname, home address, telephone number, and office number.

An additional set of standards was created in the CCITT to define electronic communications using the OSI protocol. In addition to defining the communications mechanisms, the X.400 standards also defined additional X.500 directory schema information because the X.400 mail services used the X.500 directory to store information used in addressing electronic communications.

**Standards Today**

The directory services being produced today generally fall into three categories: X.500, DNS, and LDAP. An X.500 directory server implementation is written to conform to the X.500 family of standards. There may be LDAP access mechanisms
built into the X.500 directory implementation, but the design goal of the implemen-
tation is to conform to the X.500 set of standards, which include well-defined direc-
tory protocols. X.500-compliant directory servers are offered by a number of
companies including Siemens, with its DirX product, and Critical Path, with its Glo-
bal Directory Server product.

DNS server implementations listen and respond to name daemon TCP/IP protocol
requests. Features such as dynamic DNS (DDNS) and integration with other direc-
tory services are available from different DNS server implementations.

LDAP server implementations listen and respond to lightweight directory access
protocol (LDAP) requests. Their implementation generally limits conformance to
the X.500 model to what the LDAP protocol operations require.

**X.500**

The X.500 family of standards continues to evolve. Updates to the standards were
last made in 1997. The X.500 directory data model, as defined in the X.501 standard,
serves as the basis of a number of directory service implementations. However, X.500
has not enjoyed wide acceptance as the unifying standard on which an interopera-
table, global, generic directory service could be created.

The standard suffers from describing things to such detail that existing directory ser-
vice cannot support or conform without substantial redefinition of implementation.
Almost paradoxically, the details that are critical to the success of creating an interop-
erable directory service stand in the way of X.500 directory services from becoming
the standard.

**Domain Name Service**

DNS server implementations generally follow one of two paths. The most com-
monly deployed DNS servers, which run on Unix-based operating environments,
are based on the original DNS server implementations and are shipped with Unix-
based operating systems. These DNS servers usually have a human-edited text file
containing the information served by DNS. The file is typically preprocessed into a
binary format; then a DNS server daemon process on the Unix-based machine
reads the intermediate binary format file. In-memory caching is used extensively in
DNS servers to support the high-speed host name-to-IP address translations that
are expected of it in TCP/IP networks. Host name-to-IP address translations are
optimized in DNS servers because this is the most common reason for contacting
the server.
The second form of DNS servers are those built on top of existing directory service implementations. In these implementations, the text file that contains the updateable information on which a DNS server operates is replaced with information stored and updated in a directory service. The Berkeley Internet Name Daemon (BIND) protocol is used to communicate with the DNS server, which caches information it obtains by doing initial lookups of information stored in the directory service. Implementations of DNS of this form exploit generic directory services for supporting existing, specialized, directory access protocols. The DNS server becomes a caching protocol handler, while the data-store for the DNS information exploits a generic directory service. This allows administration of DNS information to benefit from all the characteristics of content administration in these generic directory service implementations. Examples of implementations of this form are DNS services available for Novell Netware and those in Microsoft Active Directory.

DNS servers are contacted by resolvers. A resolver can be viewed as the client side of the interaction with DNS servers in a TCP/IP network. To further boost performance in interacting with DNS, caching is typically done in resolvers running on client systems, in addition to the in-memory caching done by DNS servers. The resolver is usually a service available to all users of the client system, so host name-to-IP address lookups can be done without ever leaving the resolver in many cases. The resolver on a client system is accessed using a set of well-known programming interfaces, such as gethostbyname(), which constitute most applications’ usage of DNS. The details of the BIND protocol, the server implementation, and caching are hidden for applications using it. Figure 4.5 shows the basic configuration of the DNS server and resolver functions used in accessing information in DNS.

Code Example 4.1 shows a simple IP address lookup using the gethostbyname() resolver service. The complete source for this program can be found on the CD.
Lightweight Directory Access Protocol

The term “LDAP server” has been accepted in the industry faster than implementations can be produced that support the notion. First and foremost, LDAP is an access protocol, as its name implies. The access protocol implies a data model for the information that can be accessed using the protocol. LDAP as an access protocol is defined to be carried over a TCP socket connection. Many existing directory service providers took this as a chance to create a de-facto standards-based TCP/IP connection method to their server. The industry fell in love with the notion of an access protocol definition approach to directory services, and LDAP-enabled directory services began to emerge almost overnight. Any existing directory service was a candidate for LDAP support. This has produced much confusion over just what “LDAP support” really means in a marketing publication for a product.

Since the original definition of LDAP in IETF RFC 1777 (see Appendix H), the LDAP protocol has been enhanced. The newer LDAP protocol definition, found in IETF RFCs 2251-2256 (see Appendix H), begins to address some of the difficult aspects of creating a generic distributed directory service. Some now argue that the definition of LDAP is losing its very desirable feature of being “lightweight.” Indeed, many feel that by the time everything is defined in a standard to allow for a global directory service based on the LDAP protocol, we will have everything that the X.500 standards have already defined.

LDAP is generally applied to two different forms of data. LDAP support can be applied to directories that are not generic, allowing a standards-based access method to information that is otherwise inaccessible. This is a very common mechanism employed to quickly provide LDAP support in a product. Existing system or application-specific directories can provide a path to integration of these directories by supporting LDAP access. This is a powerful tool in moving toward a directory-enabled organization.

LDAP support has also been applied to existing generic directory services implementations. The most common directory services to provide LDAP access are, not surprisingly, X.500 server implementations. LDAP support to X.500 directories is a
straightforward protocol transformation of taking LDAP requests, translating them into a subset of X.500 DAP requests, which the LDAP requests mimic, and doing the converse for results. For other generic directory services, the problem is a bit tougher because LDAP-style distinguished names must be mapped to the naming formats employed by the generic directory service. In the case of Novell NDS, this problem was simplified because the Novell NDS was originally based on the X.500 data model, lending itself naturally to LDAP access support. Novell NDS represents a directory service that has grown from its use in a LAN manager-style environment to become a generic directory service. Likewise, Active Directory represents a similar approach. The providers of Network Operating Systems have seen the benefits of directory services in providing distributed file and print services, and are building generic directory services to allow even greater distribution and management support.

A third form of LDAP support has arisen since LDAP became popular in the industry. New, generic directory servers that only accept the LDAP protocol for interaction with the server have appeared in the marketplace, such as LDAP directory servers from Netscape, Sun, and IBM. These LDAP server implementations generally use proprietary data-storage mechanisms and provide value-add characteristics in the areas of administration, extensibility, scalability, performance, or reliability. The emergence and acceptance of the LDAP protocol, however, means different vendors’ directory server implementations can be accessed by independently written LDAP clients, allowing directory servers to interoperate.

Figure 4.6 shows a set of six LDAP servers, connected by referrals, which provide a distributed tree of information. A default referral in each LDAP server is returned to LDAP clients when access requests cannot be satisfied by information contained in the server and correspond to requests for information “higher up” in the directory hierarchy. Referral objects located in entries stored in the information in LDAP server are returned to LDAP clients on requests for information “lower down” in the directory hierarchy. By connecting to any LDAP server, the client can be referred to the proper LDAP server to satisfy the request.

The introduction of LDAP has rekindled the hope that companies can build their computing infrastructure around an enterprise-wide directory service. But even with the amount of LDAP support available in the industry today, the goal of aligning an organization around a directory, from which people, places, and things across the enterprise can be defined, located, and administered, remains elusive. Directory services based on LDAP, while sophisticated in many ways, are still growing up in terms of event services, transactional support, replication, and security. Without the notions of event services and transactional support in the directory service, exploiters of LDAP-accessible directories must make compromises in the level of services they can provide.
Summary

In the past, directory service implementations have been focused and application specific, or have been applied to environments that have become unpopular in the industry today. There is a bone yard of existing directories, many of which are used extensively today in deployed applications. The popularity of LDAP and its almost universal acceptance in the industry has rekindled the vision of an enterprise organized around a distributed, reliable, scalable, directory service. The reality, however, is that the LDAP specification and the directories based on it are still evolving. The definitions of LDAP and the uses of LDAP-accessible generic directory services are really in their infancy. There is a nursery full of new directories in the marketplace.
The demands of the market to produce differentiated services will cause directories to continue to evolve and mature.

In the future, directories will play an ever-increasing role in system and service lookup, device, system, and service configuration, and user, group, and role definitions. Directories and security systems will converge to create environments in which directories are used to define, administer, look up, and modify information across the enterprise.
This section contains chapters that define the relationship between directories and the characteristics of the software solutions that make them “enterprise capable.” A book on directories, APIs, or protocols could exist without this section, but a book on enterprise software with directories could not.
Chapter 5

**MAN-RATED, ENTERPRISE-RATED, GOOD-ENOUGH, AND DON’T-CARE**

It is tempting to divide the world of software into operating systems and the programs that run under the operating systems, but this ignores a dimension of usage-based requirements and costs. Different usages require different software characteristics, which are reflected in software cost. The characteristics and costs are closely related.

In this short chapter, we develop a decidedly nonscientific usage taxonomy and apply it to directory services. We hope to show that directory services have something for every usage category, although not all categories would use a directory the same way, or to the same degree, or for the same reasons. The usage categories range from a lower end, “If it’s up, we’ll use it” (don’t-care), to an upper end, “If it’s down, someone could die” (man-rated).

**Four Classes of Usage**

Let’s define (and remember that a definition is never wrong, by definition) four basic usage classes of software. Since this book has *enterprise* in the title, it is no small wonder that business software is our key concern and the enterprise class of software usage is our focus. However, directory services can provide value to each usage class.

<table>
<thead>
<tr>
<th>Usage Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Man-rated</td>
<td>Software so important and critical that lives may depend on it</td>
</tr>
<tr>
<td>Enterprise-rated</td>
<td>Software critical to the uninterrupted operation of an enterprise</td>
</tr>
<tr>
<td>Good-enough</td>
<td>Business software, not critical but may be used frequently</td>
</tr>
<tr>
<td>Don’t-care</td>
<td>Non-critical business or personal entertainment software</td>
</tr>
</tbody>
</table>
How Much Software Is Written and at What Cost?

In general terms, *man-rated* and *don't-care* are at opposite extremes of a software usage, or criticality, spectrum. This is not the result of a study, rather the result of experience in the practical issues of software business. The exact position of the dividing lines is open to interpretation and isn’t a key part of any conclusions that we draw.

As you might imagine, the costs associated with developing software for each of the usage classes in Figure 5.1 are different. One would hope that a great deal more design and testing go into software used for air traffic control (ATC) than for the Hunt the Wumpus game (you can feel safe that this is the case for the FAA’s ATC, but we aren’t really positive about HTW).

Figure 5.2 shows that no software exists at either cost extreme, since even if someone didn’t charge for it, it still took time and resources to develop and no software costs so much that it can’t be afforded (by definition). Ignoring things like freeware and Space Shuttle control systems as extremes or special cases, we can be reasonably assured that the cost of software is directly related to the enterprise characteristics that it is required to possess (RASSS from Chapter 2, for example). If Hunt the Wumpus were man-rated, it wouldn’t be on the shelf at the local software store for $19.95, unless it sold in incredibly high volumes.

![Figure E.1](image)

*Figure E.1* Quantity of software lines written (excluding the home market)
Examples of the Four Usage Classes

Here are some examples of software systems that are probably familiar and where they fit into this usage taxonomy. Again using general terms, software cost per line of code decreases as you progress down the four usage categories listed here. “In general” ignores certain aspects of software cost such as that cost per line of code can increase with the total number of lines of code.

There is a well-documented, and for some a counter-intuitive, phenomenon whereby cost increases with complexity, which increases with the total size of the application.\(^1\) We are not concerned here with complexity issues in the four usage categories. However, using a directory instead of writing your own data management code can certainly reduce the size and hence the complexity and cost of your applications because your application will be smaller than if you have to write code to perform functions that directory servers can perform for you (such as storing data, replicating it, and providing search and sort functions).

<table>
<thead>
<tr>
<th>Usage</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Man-rated</td>
<td>Space Shuttle control</td>
</tr>
<tr>
<td></td>
<td>Air traffic control</td>
</tr>
<tr>
<td></td>
<td>911 emergency systems</td>
</tr>
<tr>
<td></td>
<td>Embedded medical device software</td>
</tr>
</tbody>
</table>

\(^1\) For example, see *The Mythical Man Month*, by Frederick P. Brooks, Addison Wesley, 1975.
To be fair, the usage category of software can depend on your point of view. If you created it, chances are it isn’t don’t-care, even if you slapped it together over a weekend. What we are after here isn’t so much quality categories as quality requirements by intended usage, because the goal is to show how directories can help satisfy the usage requirements (but the quality is still largely up to the implementers).

Also, man-rated software development is difficult for most people to grasp. For example, it is not unreasonable to require that man-rated software be absolutely frozen (no changes whatsoever) for a year prior to use in a real system. For that year, it is tested and retested in every possible configuration and with every set of runtime and environmental parameters. Elaborate procedures and even resetting of the testing time clock are required for any bug fix to be put into the code base. After it is installed, no changes can be made except for emergency situations.

Contrast man-rated software development with the speed at which consumer software comes to market, and you can see that these are truly opposite ends of a very broad spectrum. It takes a unique mindset to develop man-rated software. Knowing that someone could die as a result of your code makes you want to follow very strict procedures. It also makes you wish for provable software, but that is not currently possible in many real-world situations.

**Cost**

The only word that gets a businessperson’s attention faster than “revenue” is “cost.” After all, revenue minus cost equals bonus, or something like that. The cost associ-
ated with software comes from several related paths through the life cycle of each application.

Software cost can be broken down, again rather nonscientifically, into the following general categories. (We are ignoring the fact that these categories are treated differently for accounting purposes and do not all always apply; homegrown applications have no marketing expense, for example.)

- Market (or need) identification
- Development (R&D)
- Marketing and sales
- Life cycle
  - Configuring
  - Tuning
  - Servicing/upgrading
  - Monitoring/debugging
  - One-time charge or extended service plan

These cost categories matter because containing cost intelligently is key to productivity and growth. You wouldn’t run a car company and close all of the manufacturing facilities just to save on costs, if that prevented you from generating revenue.

Likewise, you need to know how each category of software can intelligently exploit directory services to competitive advantage. That advantage may be on a cost basis, where your software is cheaper for you to build than to buy (R&D), or it may be on a market (revenue) basis, where your software is cheaper to deploy and hence costs less to your customers in its total life cycle (and therefore wins bids against your competition). Or it may be that exploiting a directory gives your software better enterprise characteristics within a certain development time frame, which would provide advantages in both cost and revenue opportunity. Maybe you can achieve “enterprise” without much more cost than you incurred achieving “good-enough” if you can find a way to leverage someone else’s data-storing, searching, and managing software (like a directory service).

Table 5.1 offers another way to look at the general costs of software systems: development cost, cost of failure in the system, and life-cycle costs. Life-cycle costs include development cost (or purchase price), plus deployment cost and maintenance cost (debugging, monitoring, tuning, and upgrading releases).

---

2. This is a practitioner’s viewpoint. For an introduction to a more scientific approach to software cost issues, see “Evaluating the Cost of Software Quality,” Communications of the ACM, August 1998/Vol. 41, No. 8, 67–73.
Directory Exploitation

Directory services share some of the characteristics of a file system or a database. On the surface, a directory looks simply like a place to put data. Deeper down, a directory can be used to store more complex data than in a file system. Directory data can be searched in powerful ways, unlike file system data. Directories, at least standard ones, share a common representation of data (the schema syntaxes), whereas file system data are typically ASCII, or EBCDIC, or Unicode, or a specific graphic format, any of which databases or even some operating systems may not understand or handle easily. In addition, because it typically has lower system overhead, a directory service will likely require fewer hardware resources than a database or transaction processing system.

Why not just use a database? Databases take more system resources because they typically only understand 100 percent ACID data. In addition, you may have trouble modeling an organization in a relational database, whereas a hierarchical view is natural in a directory (imagine an organization as a tree, or “iron triangle” in which you want to delegate control over objects in subtree units). Directories are a natural place to put read-mostly data, hierarchical data, data whose consistency is not vital, and location information (which is data whose consistency is always suspect, whether you put it in a database, file system, or directory).

Table 5.2 shows, by usage class, some likely directory exploitation and the reasoning behind it. The “Why” column, which is over-simplified for illustration purposes,

---

3. For a brief excursion on why maintaining the ACID properties costs so much in system resources, particularly in distributed systems, see “Two Phase Commit and Recovery Logging” on your CD under this chapter.

4. An “iron triangle” is used to signify the shape (and sometimes inflexibility) of a management or organizational hierarchy. The top of the triangle would be the CEO and the bottom would be entry-level workers.
breaks down the reasons for exploitation into a small number of fundamental reasons. Many software applications exploit a directory for multiple reasons, but this rough breakdown is a good way to start looking at it. “Characteristic” means the exploitation is being done to improve some enterprise characteristic of the software (such as RASSS, defined in Chapter 2). “Cost” is development cost (R&D). “Revenue” is a measure of competitive marketability, which is possibly due to lower customer deployment cost, or life-cycle costs.

For example, a directory resident application may be easier to migrate from release to release. That is a selling point and thus is a “Revenue” item. It lowers costs for the end customer, so an application would exploit a directory this way (directory based application deployment) out of a revenue incentive, as opposed to a development-cost-reduction incentive.

**A One-Line Quiz**

Here is one line of code. If you don’t know LDAP, that’s OK, this line of code just tells the LDAP library to which server you wish to talk (it is located on dehco.com) and that you want to speak to it using the default LDAP port (LDAP_PORT=389).

```c
ldap_init("dehco.com",LDAP_PORT);
```
To which usage category would you say this code belongs? Let's assume that this is the only way the program specifies the LDAP server. The obvious problem is that it doesn’t check for an error, but assume that it does check. Now what category? Other than simple return code checking, if this line is the only place that an LDAP server is specified, then we are far, far from man-rated in this program. What happens if that server is down? What happens if the network is down? For the same reasons, this program isn’t even enterprise-rated. It might be good-enough-rated.

There are a multitude of other reasons why the program containing this line might not be man-rated or enterprise-rated, but that was enough of an example using just one line of code. Would that deficiency have come out in testing? Maybe and maybe not, depending on whether the testing included all of the environmental requirements of the program (network and LDAP server states, for example).

What could we do to improve our rating?

- Don’t rely on one server to be alive, in the same place, and running well.
- Keep a list of server replicas (defined in Chapter 19) that can be used.
- Use a locator service to find replicas (such as DNS—but what if you only have one DNS server?).
- Keep vital data in a local cache in case all connectivity is lost. You should always have enough of a local cache to avoid a total application meltdown.\(^5\)
- Check for error returns (despite the old joke that you should never test for error conditions that you can’t handle).

Truthfully, even this list won’t get you to man-rated, but it would probably get you to enterprise-rated. To get all the way up to man-rated, you wouldn’t rely on the program being able to recover from every error situation. For example, is there some strange timing problem in the LDAP library that only shows up once in a blue moon, causing the software process to fail? For sure, there will be a blue moon someday, after testing is completed and someone is on the front porch, sipping an umbrella drink under the satellite that depends on your code to maintain orbit.

To get to man-rated, you need self-restarting processes and redundant systems to account for things like once-in-a-blue-moon timing conditions, hardware failures, and accidents of a human nature (“Oops, I should have used metric, not imperial”). In fact, real man-rated systems probably would not depend on a general-purpose

---

5. Whether your application can tolerate “stale” data is a concern if you rely on a local cache. If you are keeping a local cache containing a list of servers to contact (and they all export the same function), you can tolerate stale data to a certain extent, since if you can’t contact one server, you can try another from the list. It might not matter that yet another server has advertised itself and you don’t have it in your cache, or that one of the servers in your cache is no longer available.
LDAP library, which depends on the C runtime library, which depends on the operating system, which depends on the hardware. What good is a man-rated program if the dependency chain under it isn’t man-rated?

This is why man-rated systems are hard to develop, very costly, and at a far extreme on the usage spectrum. How far business software systems go toward man-rating is a function of how important the software system is to the organization, how much cost the organization is willing to bear, and what business the organization is in. Man-rated is like excellent insurance, but did you need that much insurance?

Example Program

On the CD in the directory for this chapter you will find a sample program called locatew. The sample shows one way to locate a usable LDAP server in a business environment with a minimal amount of hard-coding or onerous user input (“Enter the IP address of the server to which you wish to connect—in binary”). Only when the program is first run does it expect to find a specific server (a bootstrapping step). The program will always use data it found in the LDAP server during its previous invocation to locate other servers in future invocations (or when a server goes down).

While this example is miles away from achieving a man-rating, the general idea is probably enterprise-rated for client systems designed for availability. Such a design might be useful for enterprise applications for roaming salespeople (not coincidentally, the case study in Chapter 23, Roaming Users with LDAP, takes advantage of this technique).

The locatew sample program has a few other interesting features that might appeal to you:

- Separate GUI and directory worker threads. This is so that the GUI never appears unresponsive when the program is accessing a directory server. This makes the programming somewhat more complicated, but the user will not be annoyed by having to wait, thinking that the program is dead. We used separate threads in this example, but we could also have used the asynchronous versions of the LDAP APIs to get a similar effect.
- A GUI and a console window. The GUI window is where the real application would do its work, like looking up products in a catalog. The console window is turned on and off from a menu item in the GUI window. The console is used to receive debugging messages, such as how the directory connection is going. There is also a menu item from the GUI window to get directory connection status, as well as a list of available directory servers. The debug console could be turned on and off with a secret key combination (handy for the people who have to service your application).
No new schema objects. We re-used an existing schema object in the Windows 2000 Active Directory meant for service publication (which is exactly what we are doing—publishing the location of directory servers).

Figure 5.3 shows the main GUI window of the locatew sample and the menu selections for retrieving directory status and available directory servers. Figure 5.4 shows the dialog that pops up when the user wants to see the available directory servers. Figure 5.5 shows the directory connection status dialog.

When locatew runs for the very first time, it attempts to bootstrap itself by connecting to a well-known server. If it is successful, it reads the locations of all other known servers from a published list and stores the list locally in a cache (in the Windows registry). If it is unsuccessful, it asks the user to specify another directory server (as seen in Figure 5.6).

Assuming locatew bootstraps its local cache of directory server locations, either by connecting the well-known server or by having the user specify a server, subsequent
invocations will work with the cache. The cache is a list of servers that advertised themselves. When locatew starts for all but the very first time, it tries to connect with the last successfully contacted server; if that fails the user is asked to select from the cached list, as shown in Figure 5.7. For good measure, the debug console is shown in Figure 5.8.

What is not covered in this example is how the locations of the directory servers were populated into a directory and kept in synchronization across the many directory servers that might be in the example organization. This cross-directory synchronization of directory data is not trivial and is covered in Chapter 20.

While an executable can be found on the CD along with all of the files for the project, it is meant to run on Windows 2000. It will not run on a system unless that system has the proper directory access client libraries installed.
Figure E.7  Cached list of servers to choose from

Figure E.8  The debug console
Summary

We defined four usage classes of software in this chapter: man-rated, enterprise-rated, good-enough, and don’t-care. Each usage class is typically associated with certain cost parameters and requires different characteristics to be present in applications. In a perfect world all software would be man-rated and free; but in the real world monetary issues get in the way (either the cost of development or the lack of revenue). We enumerated ways in which directory technology can help attack cost issues for the different usage classes of software.

What we described is a classic engineering and business problem: how to satisfy the requirements, within a certain cost envelope, within a certain time envelope, while maximizing revenue and minimizing cost. A directory service can be leveraged by software in many usage categories to speed development, lower cost, and increase revenue, because someone else wrote and tested the code to store and search many kinds of distributed data. Imagine if you had to write the operating system to get your application to run. In the future, we’ll be able to say, “Imagine you had to write the directory to get your enterprise application to run.”
A directory service must be available to various usage categories of applications, as discussed in Chapter 5. The role of an operating system may constrain what usage it can support. If an operating system is less trusted, it cannot support more stringent usage unless the provider of directory services is sufficiently ingenious. Surprisingly, there are techniques for making a service more robust than its underlying operating system. Here, we look at some operating system and robustness issues.

The Role of the Operating System

Operating systems historically provided only a management layer between applications and system resources. Today’s applications have become structured into client server components; and a directory service has been merely one more server-side application component. Some directory services may not have been classified as such until people noticed similar concepts in say, DNS, and an e-mail name-and-address book.

Meanwhile, as operating systems evolve, an increasing number of applications and support services are bundled with them. Today’s operating systems deliver out-of-the-box directory services. Unix originally didn’t deliver TCP/IP, DNS, or e-mail. Today, freely available Linux distributions contain these services and support literally hundreds of applications. Microsoft Windows didn’t originally ship with e-mail clients or networking features; Windows 98 delivers these and many more features.

In the case of LDAP directories, many Linux distributions ship with OpenLDAP, an implementation of the University of Michigan LDAP directory service. OpenLDAP contains slapd (an LDAP server), slurpd (an LDAP replication agent), and ldapd (an X.500 LDAP gateway). Microsoft’s enterprise directory, Active Directory, is probably
the most distinguishing new feature of Windows 2000. It is the enabler for many of the other prominent new features, such as group policy, Intellimirror\textsuperscript{TM} (data where the road-warrior users need it), and demand-driven installation of self-repairing software.

**Usage Ratings**

**Usage Rating of the Operating System**

What person would want his or her life to depend on the proper functioning of today’s operating systems? Think of common operating systems used today. Do any of them claim to be man-rated?

Now think about the underlying hardware. It was not designed to be fail-safe either. Current commercial hardware/software systems, as a whole, are not designed or priced to be man-rated. They are intended to carry out the yeoman work in the information age at a superb level of price-performance. Each copy of an operating system running on a given computer is not especially reliable when we consider the case where *any* failure is unacceptably costly.

Happily, redundancy and dynamic redirection, enabled by directory glue, enables groups of these operating systems and their applications to be extremely reliable to the end-user while still giving good price-performance.

**Usage Rating of the Directory Service**

A directory service is the enabler of reliability through dynamic redirection and redundancy. A directory service may maintain not only personnel information for authentication and business processes but also application service location and binding information. Now it gets interesting. A service’s location may be dynamically altered—services may physically come and go, or be moved to other network locations. A service would be administratively moved within the directory as this occurred. The binding information would be updated by a human or programmatically. A directory-enabled client or middle-tier logic would agree always to consult the directory to know where to bind to its needed services—it would never cache this information. Thus it could be unaware, say, of a failing DBMS machine’s services being redirected to another network location.

Thus, a directory service enables the possibility of an increase in the reliability of the rest of the enterprise computing services. This could potentially make the directory
service a single point of failure. We like to eliminate these in enterprises! What is a good approach toward that end?

First, multiple replicas of a directory must be scattered around a site or enterprise (see Chapter 19 for replication topics). This increases its availability in the face of network failure, operating system failure, or individual hardware failure. Each directory service site must not be an island unto itself. Next, the directory implementation’s code must be as reliable as economically possible. If a malfunction can cause corruption of an attribute and that attribute can be propagated to other directories, then our reliability glue may become unstuck. (See Figure 6.1.)

**Directory Service Users**

Clients of the directory service are not generally a potential point of failure in a distributed enterprise operation. If a user application crashes, the problem scope may be limited to the function provided by the application. If the system is reliable, the application reliability probably depends on how reliable the application itself is. If an application is rated above don’t-care it may not be the weak point for the rest of the system, unless it’s a primary, ubiquitous application such as e-mail.

![Diagram](image)

1. Business processes find a backend data connection through a nearby LDAP directory.
2. Administrator takes DB replica 1 out of service by updating entity in US-West LDAP.
3. After the LDAP replication interval expires, administrator shuts down DB replica 1.
4. Business processes find new current DB connection from directory.
5. If replication breaks, the business operation is compromised: Some processes update the wrong data or cannot find the data connection.

*Figure F.1  Problems caused by propagation of corrupted attribute*
Compensating for Low Usage Rating

There is strength in numbers. Mountaineers tie themselves together for the protection of each individual (provided that the group has a sufficient average purchase on the mountain). Redundant computer systems team up in a more socialistic manner. Here, the motivation is based on the belief that the health of the distributed applications is more important than the health of any individual computer. When operating systems are loosely lashed together through coordination supplied by directory services, the prognosis for the whole may be greater than that for the parts. Fault tolerance, redundancy, and fail-over support built into the very nature of directory services, makes this possible.

Fault Tolerance, Redundancy, and Graceful Fail-Over with Takeover

Fault tolerance may be achieved through simple resource redundancy at possible single-points of failure. (See Figure 6.2.) It is also attained through interesting cyclic redundancy algorithms such as the ECC error-correction techniques used for years in tape, main memory, or in RAID disk stacks. Here a small percentage of resource

![Diagram of potential points of failure](image)

1. The order-entry application depends on all of the DBMS systems being available at well-known locations.
2. The clients depend on the application server’s availability at a well-known location.

If any location must be changed or any component fails, business stops.

Figure F.2 Exposure to failure
redundancy—some extra bits or units—is added to the data word bit-length for use by a cyclic mathematical formula to detect and correct errors on the fly. The numbers of extra bits is a design tradeoff against how many bits may be flipped in error and still yield correct data. A failing disk drive in a RAID unit that uses ECC may be hot-swapped without a service interruption.

Error-correction techniques automatically detect failures and provide the correct data through redundancy. (See Figure 6.3.) But how does simple resource redundancy operate? How does the overall system know the failing resource? How does a new resource take over for the failing one? The answer is somewhat evasive—it varies according to the type of resource and the fail-safe architecture involved.

A “Web farm” may consist of many Web servers, all serving content from identical content pools. A TCP “sprayer” or dynamic load-balancing server assigns client requests to the least-busy web server. One or more web servers may be unplugged, but the site keeps serving content. What is it about the server that provides the load balancing? Isn’t it a potential failure point? Of course it is. That server needs a backup also. How and when does it take over? This concept seems to cascade doesn’t it?

One solution is to add services in the operating system, providing clustering support within the operating system. (See Figure 6.4.) Two or more servers share an array of disks on a common SCSI bus, and the disks are arranged in a fail-safe
RAID configuration so that one or two may go down without an interruption of service. An Ethernet or RS-232 heartbeat lifeline connects the systems participating in the cluster. A periodic “I’m here, are you here?” signal means that all is well. If one participant doesn’t send a beat, the fail-over support in the operating systems takes over the workload from the errant system even if it wasn’t aware of its failure. This process may result in a delay of five seconds or so in some clusters. The math of clustering is such that many cheaper, small computers are much more reliable than a single large, expensive computer that provides the same computing capacity.

Directories play a part in another approach, called graceful fail-over with takeover. (See Figure 6.5.) If all components of an application use a directory for location information and all agree to look in the directory for a service every time that service is needed, such services may be substituted and redirected dynamically. All that is needed is some mechanism to detect a failure, then carry out the switchover. A daemon process may monitor a service for health using an approach such as log analysis or looking for a program ID to disappear. It would conceivably update the service location information to point to another location for the service, and send an RF page or e-mail to an administrator.
The directory is not a potential single point of failure if the users and client applications can find the replicas. This brings up the conundrum of having a directory that one may use to locate everything, but not being able to locate the directory! One way to solve this problem is to use DNS as a top layer location service, and some commercial directory implementations do this. Recall from Chapter 5 that the locatew sample solved this problem without using DNS (but perhaps it required more programming).

**Summary**

Directories historically were considered an application under the operating system. The need for distributed, more nearly man-rated applications raised the importance of directories. Several modern operating systems have an integrated directory service that helps them enable new enterprise-level features.

Most operating systems are not man-rated, but it is possible to enable a good measure of fault tolerance. This is achieved through redundancy and graceful failure followed by a second system’s takeover of services. A cluster-enabled operating system enhances
overall reliability at a lower price than buying a larger, more reliable single computer. Combining operating system support for clustered fail-over and integrated directory support for directory-enabled applications provides the maximum of service possible. The man-rating of a service may be attained without any of the constituent components being man-rated.
This chapter provides more details on the users of directories. Directories are already used by just about everyone in some capacity. For example, people use directories to find Web sites (most fully qualified universal resource locators [URLs] contain a DNS hostname). Computers use directories to understand the set of users, groups, and roles defined for the system. Applications use directories to store preference and configuration information.

People

Whenever a person sits down in front of a computer to find something, a directory of some sort is involved. Probably the most common electronic distributed directory lookup used today is a DNS lookup to resolve the TCP/IP IP address corresponding to a URL. Users have become so used to these cryptic names that they’ve become part of our everyday conversation. DNS names pervade all aspects of the media, with URLs advertised over every broadcast medium.

White Pages and Yellow Pages Lookups

The phone book-style application, covered in Chapter 1 and Chapter 3, uses a directory service as the electronic equivalent of a phone book, locating people and services across an organization or group of organizations. (See Figure 7.1.)

As electronic forms of communication, business, and interaction become more and more pervasive, electronic directories of information containing up-to-date information will be used more than printed phone books. Electronic directories can be updated more quickly and thus may contain more accurate information than printed phone books. As distributed forms of security are improved, these electronic directories will allow information to be updated by the people and businesses that they reference. In business-to-business (B2B) environments, electronic directories will allow
efficient lookup of peer services between businesses. Supply chain management will use secure tunnels or secure message transport between businesses. Policy-based administration of these secure tunnel and secure message transports will be supported by information stored in the directory.

Code Example 7.1 shows how the directory would be searched using the LDAP C client APIs. Refer to Appendix A for a mini-reference on the LDAP C language programming interface as well as the definition of a C++ encapsulation of the LDAP C language interface.

People will continue to use directories as long as they need to find someone or something. As electronic forms of these directories improve in content, access, and accuracy, they will become more and more a part of everyday life. A generic directory service offers both white pages (name-based) and yellow pages (content-based) high-speed lookup and search. Imagine a distributed directory that contains information about people and businesses worldwide, allowing people to find others across the globe using a single interface. The returned information would contain physical and electronic

---

1. The use of cn, ou, l, and o for naming attributes in Figure 7.1 is discussed in Appendix A.
contact information: phone numbers, mailing and office/showroom addresses, electronic mail addresses, and World Wide Web home pages.

Operating Systems

Just about every modern operating system contains the concept of a user. From personal systems to large, multiuser systems, directories hold information about the operating characteristics of the system. Directories come in a number of forms across operating systems. In fact, much of this information is scattered across a multitude of locations, which, taken together, constitute the directory on which the operating system defines its operating characteristics.

Users and Groups

Operating systems use directories to hold user definitions. User definitions can be as simple as a table listing user ID, password, home directory, and initial program, to complex databases holding thousands of user definitions that span a homogenous set of operating system installations. These more elaborate user directories can hold information regarding user preferences, as in the Windows Registry, or application-specific information related to individual users, as in the different sets of segments found in the resource access control facility (RACF) on System 390 operating systems.

In addition to user definitions, modern, multiuser systems usually contain the notion of a group of users. Using the group concept allows us to give more than one person access to resources without having to enable individual sets of permissions. The group concept also lets us apply sets of preference information to many users without having to explicitly define these sets of preferences to each user. A related concept, which is becoming more and more prevalent, is the concept of a role. Roles can be thought of as groups associated with sets of operations, which the group is allowed to perform.

```c
int rc = LDAP_SUCCESS;
int TYPES_AND_VALUES = 0;
LDAPMessage *res = NULL;
char *attrs[] = { "mail", "telephoneNumber", NULL };
rc = ldap_search_s( ld, "l=flintrock", LDAP_SCOPE_SUBTREE,
    "(&(cn=barney*)(objectclass/inetOrgPerson))", attr,
    TYPES_AND_VALUES,
    &res );
```

Code Example G.1 Looking up a phone number and e-mail address using a directory
Apart from user and group information, operating systems also use directories to manage the set of file systems that are defined to the system. On some systems the set of file systems is defined by a lookup into a file stored in a portion of the file system that is always available. This file contains mount points as well as mount characteristics. On other operating systems the set of accessible files is defined in a central catalog. The catalog represents a directory that manages the locations of individual files. Users attempting to access files by name first contact the catalog directory and find the location of the file. Refer to Chapter 12 for in-depth coverage of related security issues in the enterprise.

**Hardware Configuration**

Another, sometimes forgotten, use of directories by operating systems has to do with hardware configuration. Many operating systems still require some form of directory for defining the set of devices that are accessible to the operating system. These devices require some set of configuration information, which is also contained in the directory that defines these devices to the operating system. The device definitions known to the operating system are usually very specific to both the hardware and the operating system. Examples of these directories are the config.sys on DOS and Windows platforms, I/O definition file (IODF) on System 390 systems, and the /dev portion of the file system on Unix-type operating systems. A generic directory service can provide unified access to this information. (See Figure 7.2.)

The code in Code Example 7.2 shows how the directory in Figure 7.2 would be searched using the LDAP C client APIs.

The code in Code Example 7.2 searches the LDAP directory starting at location `sys=sys1, ou=dusters, l=flintrock` and returns all entries that are of type (or `objectclass`) `device`, `fsinfo`, `user`, or `group`.

As operating systems come to rely on and exploit the benefits of remote administration and automated and dynamic device discovery, and to leverage user and group definitions that are common across multiple heterogeneous operating systems, their use of generic directory services will increase dramatically. When an operating system leverages a distributed, generic directory service in order to manage and define information, users benefit because true single identity/single sign-on becomes a reality, administrators benefit because remote administration as well as administration that can apply to multiple operating systems simultaneously can be performed, and applications benefit from the use of a common user and group definition usable across multiple operating environments. Operating systems will use the directory service to manage software and services, allowing distributed management of applications. Refer to Chapter 13 for a discussion of lifecycle control.
CHAPTER 7  •  DIRECTORY USERS

Linux client

LDAP server

OS

LDAP client

LDAP

dn: ou=dusters, l=flintrock

dn: l=flintrock

dn: sys=sys1, ou=dusters, l=flintrock

OS

dn: cn=user1, sys=sys1, ou=dusters, l=flintrock

dn: cn=user2, sys=sys1, ou=dusters, l=flintrock

... dn: cn=group1, cn=groups, sys=sys1, ou=dusters, l=flintrock

... dn: cn=deva, cn=devices, sys=sys1, ou=dusters, l=flintrock

... dn: cn=fs1, cn=fsys, sys=sys1, ou=dusters, l=flintrock

... 

Figure G.2  Operating system using a directory to access user, group, file system, and device information.

```c
int rc = 0;
int TYPES_AND_VALUES = 0;
LDAPMessage *res = NULL;
rc = ldap_search_s( ld, "sys=sys1, ou=dusters, l=flintrock",
    LDAP_SCOPE_SUBTREE,
    "(|(objectclass=device)(objectclass=fsinfo)(objectclass=user)(objectclass=group))",
    NULL, TYPES_AND_VALUES,
    &res);
```

Code Example G.2  Looking up user, group, file system, and device information using a directory.
The Distributed Management Task Force (DMTF), a group with representatives from many hardware and software vendors, has produced a Common Information Model (CIM). The CIM model helps to define a common set of structures on which to base directory information models. The management models in CIM will be used to enable usage of the directory service.

Networks

Today, and even more so in the future, computer networks have become almost more important than the individual computers themselves. New technologies that allow fine-grained control of network characteristics from end to end will be exploited to provide high-speed, secure connectivity between distant endpoints in a large network of computers.

Virtual private networks (VPNs) are secured connections between two networks, established across insecure communications channels, such as the Internet. Setup and management of configurations that allow for on-demand, high-speed, dedicated VPN connections require configuration of communications hardware and software at the endpoints of the connection, as well as at many intermediate nodes that will be involved for the duration of the connection. In order to coordinate all of these complex updates to multiple machines running across the network, the notion of policy-based administration was created. The idea behind policy-based administration is that all the proper individual routers and networking hardware and software configuration changes will be applied to match the overall configuration. All routers and nodes in the network must be able to access the policy information in order to create the individual configuration parameters implied by the policy.

In addition to policy-based administration, setup and maintenance of virtual private networks (VPNs), in which both endpoints of the VPN connection must have very similar configuration parameters, can benefit from using a directory to hold this shared configuration information. (See Figure 7.3.) RADIUS servers are used in many locations for providing dial-in access to a company’s internal network. A common use of a directory in conjunction with the RADIUS server is for the RADIUS server to make use of user and group definitions in the directory when it processes incoming connection requests.

The code in Code Example 7.3 shows how the directory in Figure 7.3 would be searched using the LDAP C client APIs.

As more and more network-based services and applications are established, the need for consistent and coordinated configuration information across multiple systems
will become a requirement. Using a generic directory service can help solve the consistency and coordination issues in configuring network services.

**Applications**

In a complex enterprise environment, applications can use directories in many ways. Applications almost always require some set of configuration information in order to run the application once it has been installed in the environment. In many cases,
applications must locate and contact other applications running on other systems. Applications use directories to locate and contact other systems and software. Applications usually require user definitions and sometimes group definitions associated with users. All of these uses lend themselves to solutions that incorporate a generic directory service for information storage and retrieval.

**Configuration**

Applications may define their configuration in a set of configuration or “side” files. In these cases, administration of this information becomes the major administrative task of the application. Providing a command-line or graphical user interface (GUI) for this configuration information is a development expense and necessarily implies a separate set of interfaces, wizards, and management tools to support the configuration. When applications use a generic directory service to store their configuration information, the interfaces are defined and the administrative user interface can be consolidated into a larger, cross-product, consistent, distributed administration “console.” While this can be done without using a generic directory service to hold the configuration, the implementation is simplified if updates to the directory are performed instead of requiring updates to remote configuration files. See Chapter 28 for a case study involving the use of a directory to manage the configuration of a Web farm.

As distributed middleware becomes more and more complex, a generic directory service can help ease the complexity by providing a centrally administered, widely accessible information source for configuration information. Applications such as IBM’s CICS, MQSeries, DB2, and Lotus Domino have large requirements on distributed administration.

**Location**

In most cases, applications that communicate with other applications must determine the network access addresses and protocols to be used to make the connection. These applications use directories to locate remote computers and software. An example of this is the Object Management Group (OMG) Name Service definition. The OMG Name Service provides a name to distributed object instance mapping that is used to build client-side proxy object instances through which the distributed object can be contacted and used. The OMG Name Service provides a directory of information pertaining to the distributed objects that are accessible by name in the directory name space. Another example of an application that uses a directory to locate and contact remote systems and services is the generic Web browser. Web browsers contact DNS servers to resolve host names to IP addresses so that information can be loaded into the browser from the remote location.
Users and Groups

It seems that for every application available in environments today there are user and group definitions stored and managed by the application. Electronic commerce web applications, single sign-on solutions, automated backup applications, and electronic mail and groupware applications, all define and exploit user and group information stored in directories. Administration of each of these directories usually brings with it a separate set of programmable and graphical interfaces. The definition of a user becomes scattered across a multitude of directories so that modifications, in some cases, must be performed multiple times, using each applications’ set of interfaces. A generic directory service can provide a single point of administration of this information. (See Figure 7.4.)

The code in Code Example 7.4 shows how the directory in Figure 7.4 would be searched using the LDAP C client APIs.

Applications will begin to adopt a generic directory service for storing and retrieving information. Applications that store and retrieve their configuration information in a generic directory service can benefit from using a generic interface to access and update this information. Applications can use a generic directory service for retrieving

![Diagram](image_url)
Remote server and application locations. Applications using a generic directory service for user and group definitions can benefit from a common user definition across multiple applications. Ultimately this simplifies administration for users and groups and provides single sign-on to end users across multiple applications.

Summary

Individual users, computers, networks, and applications use directories. Each can benefit from storing and accessing information in a generic directory service instead of in various files and databases scattered across network-connected systems. Using a generic directory service can reduce or eliminate duplication of information across applications and can simplify administration and usage of multiple applications.
Chapter 8

THE EVOLUTION OF APPLICATION MODELS

“Dave, my mind is going! I can feel it! I can feel it!”
—2001: A SPACE ODYSSEY

Application models and computers at the beginning of the new millennium have not quite evolved to the level of HAL in 2001: A Space Odyssey, although evolution has occurred rapidly. Pioneer programmers were happy just to get a computer to do something interesting before it malfunctioned. The evolution of application models followed the evolution of computers and operating systems. A directory service can enable better enterprise characteristics for appreciation models.

Enterprise Application Models

The concept of an application model is subjective. Let’s look at some network application models and glue technologies that pertain to enterprise applications.

Monolithic Applications

The logic in a monolithic application is largely contained in one address space of one computer. Monolithic applications, such as Microsoft Office 2000 and Lotus SmartSuite are common today, especially in desktop PCs.

At first glance, it may seem that such an application could not be monolithic if it used a directory service. Now visualize an enterprise with hundreds of users for each of these applications. The words “deployment,” “maintenance,” “authentication,” “authorization,” “policy,” and “administration” may come into mind. A directory service may form the underpinnings of a centralized infrastructure that facilitates all or some of these tasks.

Windows 2000, for example, uses its Active Directory to enable some of those characteristics. End users are authenticated through the directory service and an application
may be registered in the directory service. This application is offered to end users who are authorized within the directory service to use it. An application may even be forced to such a user's desktop based on policy rules residing in the directory service. Certain features may be turned off for certain users by policy rules. Such an application may be upgraded remotely using system facilities that maintain state in the directory. The application is still monolithic. It runs self-contained in one address space, but the state of its file bits, and who can use what parts of it where, are kept in a distributed directory service.

**Client-Server Applications**

A huge amount of enterprise data landed in database management systems (DBMS) during the 1970s and 1980s. This data was often manipulated through batch processing systems. Orders or updates were keyed into update streams that were periodically applied as updates to the database. Queries or batch report programs generated huge binders of paper reports that were manually referenced. This is still a part of the data processing model in some businesses today.

The amount of data grew over time. The DBMS became accessible over a network. The clients freely dispersed to other machines. These new applications were not monolithic. They were the first of the client-server strain. Here, many copies of one application, running in several computers, all shared one remote copy of the database. The applications were clients of the DBMS, called a **database server**.

The clients may leverage a directory service in the ways discussed for monolithic applications. The monolithic client talks to a remote DBMS to form the client-server application. How does the client locate that DBMS? It could consult a DNS server to map a network name to an IP address. A DNS server provides a distributed directory service. The DNS server could recursively chase referrals to an authoritative DNS server to carry this out. Alternatively, the client could consult a service connection point in an LDAP directory to locate the network location of the DBMS. Each of these approaches involves a form of directory service.

How does an administrator move the DBMS to another server without interrupting the clients? An LDAP-based service locator works well if the client always uses it without caching the DBMS location. The administrator may relocate the server by bringing up a DBMS replica and updating the connection point in the directory service. This doesn’t work well with DNS because its client-side resolver may preserve the old DBMS location in its cache for a time.

**Thick Client Applications**

Client business logic and presentation logic that reference a backend network DBMS is called a **thick client application**. (See Figure 8.1.) This seems to be a good bargain:
Logic and presentation cycles are off-loaded to a lot of cheap desktop machines that leverage a huge centralized data store. But there are two problems:

- Increased maintenance and deployment costs
- Heavy client and network resource use

Thick client applications often need increased deployment and maintenance effort. How is the current version of the client to be deployed to the users? What happens when a change to the server requires a version update of the client? If the new version is not downward compatible, all users will require a new version of the client simultaneously at the instant the server is upgraded.

Multiple databases on dispersed servers may be joined in some client operations. These could be resource-intensive operations. The contents of entire database tables could flow across the wire to many clients, contributing to network congestion. Each client must buffer table data as it operates on it.

A directory service may be leveraged as with monolithic and client-server applications. It may especially ease the deployment and maintenance problems of the thick clients.

**Thin Client Applications**

One attraction of client-server architecture was the promise of offloading resource requirements from a small desktop computer to a larger, shared central server, but only the data was offloaded in the thick client model. A modern DBMS may store execution logic procedures in its own data. The thin client model (see Figure 8.2) offloads...
procedural logic to the DBMS. Tables do not have to be transmitted over the wire and clients do not need to buffer the tables. Only the input and final results are transmitted. The client part of the application consists of a user interface that gathers and validates form data.

One disadvantage of thin client applications was that the user interface might not be as rich as that of a thick client appreciation. This is partly because of the desire to reduce round-trips to the server and to keep the client simple so that it would be relatively independent of the version of the server-side of the application. A directory service can enable the clients to locate servers dynamically, no matter where they may reside.

**Tiered Applications**

The deployment and maintenance costs of client-server applications were a problem. The emergence of the World Wide Web, Java, and Microsoft ActiveX controls caused people to take a second look at client-server application architecture and application deployment. At first the Web simply projected HTML files from a server to client Web browsers, where the files were formatted and displayed as static brochures. The Web server was soon leveraged to execute client requests in a stateless script, sending generated HTML results back to the client. One server application extension is called common gateway interface (CGI). Splitting an application into networked parts leads to the notion of tiered applications. Here, the user interface, the data, and at least some of the logic reside in separate processes or machines.
Two-Tier Web Applications  A Web browser may receive an HTML page that contains an input form to a server-side CGI program, a script, or a Java servlet. An application may consist of several server-browser form exchanges. As we move slightly up the food chain, an HTML page may contain script or binary components that execute in the Web browser. These may converse with the server side of the application, and directly with stored procedure logic in a separate DBMS using a separate wire protocol. If the client logic were to locate the DBMS through a directory service, the DBMS machine could be dynamically moved, as discussed before.

The advantage of two-tier applications is the ease of design and implementation. A disadvantage of the pre-Web versions was difficulty of mass deployment and maintenance. The Web-based variants are a boon to the client-server paradigm. An HTML page deploys the correct application version. Updates synchronized with the DBMS may be deployed instantaneously, particularly if a DBMS were to be located through a directory service.

A lingering disadvantage of two-tier applications in an era of larger and larger environments is that of scalability. Huge numbers of client requests may cause an early performance plateau on an enterprise-level application.

Three-Tier Web Applications  Further partitioning of the application addressed scalability problems. This brings us to today's three-tier application model, in which the deployment mechanism is restricted to Web-based Intranet or Internet applications. The Web server becomes, or cooperates with, an application server. The client gathers and validates form data. Middle-tier business logic is contained in the application server. The business logic carries out third-tier DBMS queries on behalf of the client, instead of allowing the client to do the requests directly, as in the two-tier model. The middle tier is a proxy client of the DBMS or data repository, on behalf of the remote client. Such applications may be more complex to develop and debug than two-tier or monolithic applications, but they usually scale better because the logic and database connections in the middle tier may be re-used across multitudes of mostly dormant (in computer time) clients. It's like a few trunk lines—the DBMS connections—being shared by many more office phones—the clients—through a PBX—the application server.

All three tiers may utilize a directory service. Tier one may use the distributed miracle of a DNS server to locate the correct application server. The middle tier could dynamically locate the tier-three DBMS through a directory service, as usual. Additionally, the middle tier may authenticate user requests through a directory service instead of requiring the DBMS to authenticate each request. That way DBMS only authenticates the middle tier, so that pooled, reusable, open DBMS connections may be pooled for one user—the middle tier. Finally, the third tier could be a distributed database keeping its configuration in a directory service. (See Figure 8.3.)
N-Tier Web Applications  Three-tier applications are generalized to $n$-tier applications. Dynamic load balancing across server machines and farms of identical application servers create many tiers. This is the distributed application model in vogue today, within enterprises and in e-Business. A directory service may be used in configuration management, service location, and authentication in this application model.

Client-Server Glue Technologies

Some of the technologies and patterns used in the design of thick, thin, and tiered applications are often combined within one client-server application.

Transactions

Most serious client-server technologies involve changing some data or viewing the data with a set of clients while it is subject to change by other clients. Today’s e-Business applications would fall apart without a mechanism to ensure data integrity. A full-blown distributed transaction service is employed by the business logic to enforce the classic ACID transaction properties discussed in Chapter 2. The transaction service adds another software server that may be managed with the aid of a directory service in an $n$-tier client-server application.

Message-Oriented Middleware

Each request from a client to a server may be queued instead of being carried out as a direct request or remote procedure call. Because every part of the application
doesn’t have to be online at the same time, this increases reliability, flexibility, and interoperability.

The developers of message-oriented distributed applications employ message-oriented middleware (MOM), another type of system services. Message queuing may be used to glue legacy systems on disparate platforms into one distributed application, complete with an umbrella of transaction protection. Thus a distributed collection of CORBA, EJB, and COM+ processes could interoperate at the message level with a transaction. A directory service may be used to manage the location and configuration of message queues. Queue users and client services may be authenticated through a directory.

**Boss-Worker Thread Model**

Servers usually consist of pools of threads. A **boss** thread listens on a communications port for inbound requests from clients. When a request is received, the boss thread assigns the request to a **worker** thread from a pool of dormant threads. The worker carries out the request and notifies the client of the result. The boss may maintain this pool by creating or deleting worker threads, balancing system resource re-use and the time needed to create threads. The relatively long time needed to create a thread is amortized by re-use across many independent client requests. When such a thread finishes its work, it notifies the boss thread, which, in turn, replaces the thread in the pool of available thread. (See Figure 8.4.)

![Figure H.4 Boss-worker model](image-url)
A directory service may aid in remote configuration management of the server. The operating parameters may be remotely configured and one or more directory-enabled boss-worker servers would listen for updates.

Web servers and application servers use this model to achieve high overall application throughput. Microsoft COM+ and MTS use this model to optionally pool stateless COM business objects. As a result, the developer can create complex transactions in a simple-to-implement manner.

**Managed Components**

The server-side scripts usually invoke stateless, reusable COM logic or Java servlet logic. These involve use of components—reusable packages of logic having well-defined interfaces. A component instance manager carries out a component instance management function that promotes scalability by preventing numbers of instances of components from tying up server resources. (See Figure 8.5.) A side benefit is declarative ACID distributed transaction protection. Access to these components may be controlled by a directory authentication mechanism.

---

**Figure H.5** A component manager

- Client “creates” instance over the wire (expensive)—manager gives it a stub instead.
- Client enters a Tx (expensive)—manager lends it a stateless instance (cheap reuse).
- Client passes database ID to instance.
- Instance sets its state from DBMS read (expensive, but need DB read anyway).
- Processes and updates DB(s).
- Ends Tx—instance either destroyed or pooled depending on configuration.
- Client retains proxy-stub connection but the instance is gone.
Applications on Symmetric Multiple Processors

Modern super-server machines contain several processing units. Such tight coupling is called multiprocessing. If the processors and their busses are identical, they form a symmetrical multiprocessing (SMP) machine. SMP is achieved with intimate assistance from the operating system. Good server applications may be optimized to take advantage of tight coupling. The throughput of a server on an SMP machine may actually drop below its throughput on a single processor because of resource contention.

Clustering

Several distinct computer boxes may be grouped such that they appear as one loosely coupled cluster. There are two types of clustering:

- Workload balancing across several machines
- Redundant fail-over in the event of a machine failure, as mentioned in Chapter 6.

Either of these features requires substantial assistance from the operating system.

Workload Balancing

A middle-tier server application may be installed on several machines on a workload-balanced cluster. A dispatcher sends each request to the historically least-busy machine for processing. A variation is a Web farm—a group of Web servers or application servers that appear to the client as one URL. Chapter 28 presents a working example of a small Linux Web farm that locates its backend DBMS through a directory service. The advantages of load balancing are flexibility, better processor utilization, and the ability to add processor cycles merely by adding machines to the workload cluster. Some form of directory service would aid in managing this task.

Fail-Over Clusters

Server reliability may be achieved by buying a very reliable server, but this can be expensive and part failure and environmental problems are still a risk. Reliability can be achieved instead by grouping relatively cheap, more failure-prone machines in a loose cluster. The cluster members share electrical paths to an array of disk drives,
which themselves may be arranged in a RAID cluster to provide data redundancy or error correction in case of a drive failure.

Each processor communicates with its peers through a “I’m in good health” heart-beat signal through a simple network cable. If a heartbeat is missed, the operating systems on the rest of the machines grab resources from the failing machine. A client user may notice a five-second response and somewhat slower than normal response times afterward because one processor is missing.

The advantage of fail-over clustering, combined with workload balancing, is that several cheap, unreliable machines may equal the processing power of a larger reliable machine while actually exceeding its reliability. Figure 8.6 illustrates that if four inexpensive machines join a fail-over cluster, the reliability of the cheap cluster, as a whole, is an outage about every 10,000 days, or 27 years!

**Summary**

Applications began in primitive computers as single monolithic images. As computers and networks evolved, applications broke into separate pieces joined by a network and a multitude of client-server network-distributed applications arose. The advent of the Web saw the three-tier or n-tier model become popular. Many glue technologies, such as DBMS and transaction monitors, arose to enable e-Business and enterprise-level distributed applications. Scalability and reliability needs started to be met with SMP and clustering.
Chapter 9  
**AVAILABILITY**

Availability is the measure of how much time your application or system spends doing useful work versus how much time it spends down. Availability is not the same as reliability. Reliability is a measure of failure rates; availability is concerned with recovering from failure and continuing. An application can be full of bugs, but if it recovers from these quickly, it can still be highly available. On the other hand, an application can be completely bug-free but still not be 100 percent available if, for example, it is impossible to migrate from one release of the application to another without taking the business down for a month.

Availability is a key enterprise software characteristic. Directories must be highly available before enterprises can rely on them in building enterprise applications. In addition, if you have a highly available directory service, you can improve the availability of the applications using it.

### 24×7 and 24×365

24×7 refers to 24 hours a day, 7 days a week. It is sometimes used to refer to systems that must be highly available. Real business, of course, runs 24×365. How available do real business systems have to be? If the business depends on it, then the system must be available whenever the business is running. If the business is a multinational corporation, the system is probably running just about all of the time.

One of the authors first learned about real-world availability when discussing a new operating system feature with a customer in Tokyo in 1988. The author thought the customer would be upset to learn that it would take 12 months to get a new operating system feature and would tear our heads off for being slow. “No,” they said, “We want it in eighteen months.” The customer wanted the feature and in 18 months because the next service window for the application that needed the feature was about two years away. A service window is an opportunity, while an application is down, to do maintenance. The customer’s application had been running nonstop for two years already with no outages. That’s an example of enterprise-level availability.
Today, that customer would probably have a test system and apply the new feature to the test system in a pilot rollout, because business expects to add new features to its customer service systems faster than in the past, but with the same degree of total-system availability—that is, $24 \times 365$.

**Servers and Services**

A server, whether software or hardware, is a single logical entity. A software server runs in a software process and has at least one thread of execution. A hardware server, unless it is clustered, is thought of as a single box. There are many techniques that a software or hardware server can use to increase its resilience to failure and therefore increase its availability. Software servers can isolate work into units that can fail without affecting the software process. Hardware servers can isolate failing redundant parts using software or firmware, and switch over to the nonfailing components (and naturally “phone home” to schedule a service visit). These methods (isolation and redundancy) are not so different between hardware and software servers, and are applicable to directories and to your applications as well.

A multiprocess software service is the software equivalent of hardware clustering. A service is a logical entity that can, and usually will, run on several hardware boxes simultaneously. Think about a DNS that only ran on a single hardware server. Of course it doesn’t run on one box, it runs on thousands of them within the context of the Internet. That is because DNS needs to be highly available. Figure 9.1 shows a directory server running on one hardware box and a directory service running on multiple hardware boxes. Not shown is that if the client of a service loses its connection to a particular server, it can reconnect to another server.

Creating a software service is considerably more difficult than creating a software server. Everything that you do to design a server also applies to designing a service, plus you must coordinate your service across multiple running instances of itself. Each server that makes up a service is just one piece, but there are multiple pieces to the whole.

Since it is harder to create a service than to create a server, why not just create one big server and make it man-rated? There are several reasons why you wouldn’t necessarily want to do that, including, as we know from Chapter 5, that man-rating would require redundancy anyway. The other reasons for creating a service out of multiple servers include the following.

Performance—if there are several servers in the service, they can be placed strategically to provide better response times to the clients.

Scalability—an architecture of cooperating servers can make it easier to add capacity incrementally when the need arises.
Serviceability—with many servers, you can debug one while leaving others up and running. You can also try out new versions of the service by migrating just one “test” server, or rolling out a new release slowly across the servers.

Availability—since there are multiple servers in the service, they will all have to experience a failure before the service itself is totally unavailable.

The last item in the list, availability, applies to both directories and the applications that use directories. By examining how a directory provides a highly available service, you will see how to use a directory service to build highly available applications.

### High Availability

Availability of a service can be achieved several ways, including robust server code and redundant servers.

### Robust Server Code

Robust code is like motherhood and apple pie; that is, you can’t argue against robustness. However, there are various flavors of robustness that apply to any server. How does an operating system maintain a level of robustness? After all, an operating system
is just a general-purpose server, or a server on which to run other software servers, such as directories.

Many modern operating systems employ a hierarchy of isolation to achieve robust behavior. For some, it is a deep hierarchy and for others it is shallower. For example, the OS/390 operating system isolates batch work into a multilayered protection scheme. Each layer in this scheme has its own unique error detection and correction logic. The more trustworthy the code is, the less isolation and fewer levels of the isolation hierarchy are needed. However, even the most trustworthy code in the system (the operating system’s innermost kernel) is under special error detecting and correcting watchers. Even the hardware, through its microcode, participates in this scheme in several ways. Figure 9.2 shows the various levels used in the OS/390 scheme for making the batch service highly available.

As shown in Figure 9.2, one of the ways to create a highly available system is to assume that failures will happen in increasing frequency up the vertical axis, which also indicates decreasing trustworthiness. This allows designers to isolate failures at each level and also shows them where the system must be the most “bullet proof,” since every layer simply relies on the layer below it to catch and handle errors. Figure 9.2 shows how to increase availability of the OS/390 batch facility.

- An initiator doesn’t depend on the reliability of a job step to maintain the availability of the process to run batch jobs.
- The batch service (the JES) doesn’t depend on one initiator to maintain the availability of the batch service. There are multiple initiators for several reasons, among them availability of initiator services to run batch jobs.
- The master scheduler doesn’t depend on the JES to service other types of work (interactive work, for example).

![Figure 9.2](image)

**Figure 9.2** The wedding cake of software availability / reliability
The kernel can restart the master scheduler, if needed, to maintain the availability of the system as a whole to run different kinds of work.

A layer not shown is that several OS/390 systems can cooperate in tightly coupled sysplex of systems for even higher availability and a different level of failure detection and correction. This is the redundant server approach.

Every level in Figure 9.2 uses different mechanisms for catching and correcting problems. For example, every initiator runs in a separate software process, which provides a level of hardware and software isolation. Some of these levels depend on special-purpose features built into the hardware and microcode (OS/390 and its S/390 hardware grew up together over many years in a quest for ultimate reliability).

In Figure 9.2, what happens if a failure occurs (and is not handled) in the job step, which is untrusted code written by any application writer? The layer below it, in this case the initiator, detects the failure, reports it, and flushes out the job to make way for another one. The same thing happens down the hierarchy, using increasingly sophisticated levels of detection and correction and with different levels of severity of action.

A directory service can take advantage of this kind of availability hierarchy in several ways. If the directory service is a part of the trusted base of the operating system, it can integrate into the system's availability hierarchy at a deep level. However, if the directory is simply a job step, as in Figure 9.2, the options for integrating into the availability hierarchy can be more limited. However, even if it doesn't integrate well into the availability hierarchy of the operating system, the directory server can still be built emulating this levels-of-recovery scheme within itself, and so can your application code.

Your applications can emulate levels-of-recovery scheme, for example, by accessing a directory server on a worker thread. If there happens to be an unrecoverable error, you can always have another thread (the boss, or the dispatcher, or other term for Beelzebub) kill it and start a replacement thread. Of course this is just one level. Another lower (or higher depending on your vantage point) level could be a sister process that watches the real process and restarts it if it dies.

**Redundant Servers**

An entirely different way to increase the availability of a service is by using redundant servers. If you have two identical software servers running in different boxes and one of them dies, you can still provide the service to clients, although with degraded performance characteristics. It doesn't matter whether the error that caused the server to fail was a hardware malfunction, a software bug, or an errant human action (tripping over the power cord, shutting the system down, or, the perennial favorite, hitting the large red emergency power off switch located in every data center. If the redundant servers are located in physically distant places, the service should continue.
Redundancy can be built into servers several ways. One way, which is easiest to implement, anoints one server as the king and all other servers as subservient. For example, only one server will contain the read-write copy of the directory data and all other servers exist as read-only copies. This single-master method is fine for increasing read-only availability, but it doesn't increase the read-write access availability. A multimaster scheme is one in which every server contains a read-write copy of the data. Multimaster is harder to implement than single-master. (See Figure 9.3.)

The redundant servers and robust code methods are not mutually exclusive. Depending on where you aspire to be on the scale of enterprise characteristics (for example, the RASSS scale), you may need a measure of both methods. This is particularly true if you are building an Internet-based service and the viability of your business depends on the availability of your service (for example, an on-line stock brokerage, or a Web storefront). Redundant directory servers, or replicas, are discussed in more detail in Chapter 19.

Low Availability

Any failure that results in the loss of a service causes availability to drop. The following are causes of availability problems.

- A service built using a single software server where the server has any form or bug or error condition from which it cannot recover, or the underlying operating system, communication layer, or hardware has such a condition.
Even a memory leak of one byte per directory server request will eventually bring the server down and adversely effect availability. This slow memory leak will cause a problem only when high volumes of requests occur for extended periods. Applying Murphy’s Law, the server will come down when in production on a high-volume day.

- A worldwide cataclysm when using a service built with multiple, physically distant software servers. Note that, barring a cataclysm, a slow memory leak that brings down one server at a time probably will not affect the availability of such a service.

Intuitively, it should be easier to create redundancy than to eliminate every possible source of uncorrectable errors. Suppose a software system exhibits 99 percent availability (it is up only 99 percent of the time)—99 percent is really bad, by the way. If you have two such servers cooperating in a service, and each server is available 99 percent of the time, total unavailability of the service would occur only when both servers are down. Sudden, the availability of the service might be acceptable. This ignores some other important factors, like what happens to the work in progress when one of these 99 percent servers suffers a failure, or whether the 1 percent is caused by something in the cooperation logic of these servers so that when one goes down the other exhibits “sympathy sickness.”

Figure 9.4 shows that many availability-robbing bugs can be eliminated, but eliminating the last few is the hardest and therefore most expensive. This can be observed in

![Figure 9.4](image)

**Figure 9.4** Cost of removing the last few bugs

1. See Chapter 8 for availability formulas of multiple systems.
software engineering practice, although it may change in the future. Wiping out the last vestiges of every possible unhandled error condition is extremely expensive and still does not render the program provably correct in systems of any real-world complexity.

Figure 9.5 shows, in very general terms, the design and development cost of adding more servers to make up a software service. There is a large jump in cost from one to two servers, because the design must change drastically when cooperating servers interact. However, once the cost of making the service fundamentally distributed with multiple servers has been borne, it is less expensive to take that design and extend it to many servers.

**Improving Application Availability**

How do you take advantage of directory services to increase the availability of your service or application? The canonical example for improving the availability of any server application (any application to which clients connect to get something done) uses location services.

Suppose you write a server to which clients connect and make requests. This server location is hard-coded into the client software or in a file on the client machine.
That’s bad if the server needs to move, or you want to add load balancing across servers, or a server is simply down.

The solution is simple. Advertise your server services in a directory service. When the client software comes up, it connects to the directory service and asks where the service is. It could get back a list of the currently running servers to which it can connect.

This improves availability by making the location and state of the servers independent of your client application coding. That is, you can add new servers, they could move, and they can even add new features that are also advertised in the directory so that clients can take advantage of them once the client software is upgraded. If one of these servers goes down, your client code can find another one using the directory. In short, if the service is modeled in the directory, you can get availability benefits for your application.

Think of the classic model-view-controller concept from object-oriented programming. The model is data. The view shows aspects of the model to users. The controller operates on the data, at the behest of users who are probably looking at the view(s). Now, what is a directory service? It is a data storage and retrieval mechanism. You just need to add a controller and possibly a view to this very convenient holder of models (data) to enable a higher availability solution to whatever your problem happens to be.

**Advertising and Finding Services**

In this chapter’s directory on the CD, you will find two simple LDAP programs, one called `client` and the other called `server`. The client code shows how a client can locate servers using directory services. The server code is the shell of a server application that advertises its services in the directory.

The client application sample code is completely decoupled from information about the location of server applications. Instead of hard-coding the location, the client uses the directory service to find servers and information about them. This can increase availability of both the client and server programs by making them much more flexible.

- Clients can find servers wherever they move. Client software does not have to keep error-prone side files or ask human users for help.
- Servers can coordinate how many instances are available for client connection without regard to clients having problems finding them (this part is beyond the scope of our example).

The example client and server code could be much more sophisticated. The program does randomly choose among a list of advertised servers (a very primitive but effective form of load balancing), but it could also publish service data (such as information...
about the specific interfaces that the server exports) and performance data so clients
could make intelligent choices about which server to contact. The example code is in
some ways just a generalized version of the locatew example in Chapter 5. In locatew,
we looked at how to use a directory and local cache in a coordinated way to keep con-
nection information for directory servers. In the client and server code, we look at
generic aspects of advertising and locating any kind of server. A good enterprise appli-
cation would incorporate all of the features of locatew, client, and server to achieve
availability and flexibility.

Locatew (Chapter 5) shows how to maintain a cache of advertised directory servers,
so that directory services are always available (whether to a client or another
server).

Server shows how to advertise a server (whether directory server, application
server, database server, or other kind of server) with very simple LDAP calls.
This enables servers to move, if necessary, and clients don’t have to hard-code
the locations of servers.

Client shows how a client can locate servers using very simple LDAP calls. Clients
locating servers using a directory are flexible to changes in server location and can
even make intelligent decisions about which server to use.

The client and server programs are simple Windows console applications (without a
GUI). Figures 9.6 and 9.7 show the client and server programs running in the same

![Client console](image)
system and on the same desktop, but of course they could run anywhere in a network with access to directory services.

When the server program comes up, it advertises its IP and port information in an attribute called serviceBindingInformation which is in serviceConnectionPoint (SCP) directory objects. These are predefined objects in the Windows 2000 schema. Server also puts its IP and port information to the screen, as can be seen in Figure 9.7.

When the client program comes up, it searches the directory for relevant SCPs, filtering them by a naming convention to find only server advertisements of interest. The program will attempt to connect a socket between itself and a server. If it is successful in both finding a server through the directory and connecting to the server, the client program sends messages from the user typing at the keyboard to the server, as can be seen in Figure 9.6.

**Beyond Location Information**

Don’t stop with location data in a directory. You can think up many more ways to use a directory service to make client and server software more available, flexible, rich, and better performing. For example, why not make the client software look itself up
in the directory? It can make sure that it is running with the latest fixes, or at the latest release. This improves your application availability: The application can update itself so you don’t have to do it manually for all 10,000 workstations. The application doesn’t need to be unavailable just because you are migrating to a new release or applying fixes. Servers can likewise look themselves up. This is an example of client or server configuration and life cycle management, and is covered in Chapter 13, Life Cycle Control, and Chapter 26, Application Management.

Summary

Availability is not the same as reliability, but they are both related to how well-behaved an application is. Availability is the “A” in the RASSS.

Directory services themselves can be made highly available by using a variety of techniques, ranging from brute code quality through redundant servers. Directory services can also be used to make your applications more available and hence enable your business to run 24 × 365.

To illustrate client and server availability improvements, we presented two related programming examples called client and server.
Chapter 10
SCALING

“If we cannot maintain the scalability, speed and security of our network, customers will not accept our services.”
—FROM A COMPANY’S QUARTERLY REPORT

Scalability means an application is able to absorb additional work without undue degradation of its performance characteristics. While scalability is often measured in terms of transactions per second, or bytes of data processed, this chapter focuses on the number of users supported. Enterprise software needs to scale from small numbers of users with small amounts of data to the largest extremes. A small deployment might involve a workgroup with tens of employees, while the high end covers worldwide access from tens of millions of enterprise customers.

Scalability

How scalable does an application system need to be? That depends on how many people will be using the application, and how much load each will place on it. For many emerging Internet e-Business applications, the desired answer is everyone! Multiple commercial sites already support tens of millions of users every day. But every enterprise application will not need to support the entire range as shown in Figure 10.1.

At the low end of scalability are applications developed for the small office or home environment. These applications need to support a very small number of users and many have been designed for a single user. For example, most retail software stores a single set of user preferences. However, even home users are starting to require applications that support multiple users. A normal home computer may be used by multiple family members, each of whom wants his or her own application preferences. Aside from the issues related to supporting multiple sequential users, most home and personal productivity applications do not have major scalability concerns.

Next up the scalability requirement are applications designed to support workgroups—a set of people working on a common project. Workgroup applications are designed to support the interactions of multiple people against a set of data. The data
might be as simple as a group’s conference room calendar or as complex as the architectural drawings for a new building. The key difference between applications designed for workgroups and those designed for single users is the requirement for easy access to common data by many people. Workgroups introduce the first real scalability issues. For example, can the system support two people accessing the data at the same time? What happens when another person joins the project—can the application handle the additional workload?

Many enterprise business systems are designed to support more than a single workgroup. Applications like time cards, calendars, expense accounting, and e-mail have more utility and value when they can support an entire enterprise or at least the major divisions or locations of an enterprise. These intra-company applications need to scale based on the number of employees within the company. Compared to workgroup applications, intra-company applications have to support more users and significantly more data. While a workgroup e-mail system may need to support tens to hundreds of users, the e-mail system for a major corporation may need to handle hundreds of thousands of users.

Just a few years ago, the users of an enterprise’s applications tended to be employees only. Today, with the rapid growth of the Internet, many enterprises are discovering

![Figure J.1 Scaling the number of users](image-url)
the business value (either in cost reductions or revenue generation) of making internal systems available to nonemployees. This ranges from tighter interconnection with business partners to on-line support and marketing to customers.

Planning Ahead

Many people assume that applications designed for a workgroup will never need to scale to the enterprise or beyond. They are often sadly mistaken. Valuable applications take on a life of their own and often get redeployed to additional workgroups. Then comes the time when two workgroups want to share the application and its data. The next thing you know, a workgroup application meant for a small number of people is critical to the daily business of the enterprise. As this enterprise-level usage grows, people find that the workgroup application does not have the required level of enterprise software characteristics. One can view this as the “buried under success principal.”

One workgroup application that has grown into an enterprise applications is the Web browser and server. These applications were originally created to enable a workgroup (particle physicists at CERN) to easily share experimental data. Today the Web browser and URLs have become ubiquitous. While the Web browser only needs to scale to a few users, Web services exist that require scaling far beyond the confines of a single enterprise. For example, in June 1999, eBay had over 5 million users and amazon.com had over 10 million users.

Another application that has grown up from the workgroup environment is the collaborative computing system. Collaborative systems started as application that enabled workgroup members to work on a single document together. Lotus Notes, one such application, has grown into an enterprise collaboration system, enabling companies to extend collaboration and workflow processing across the entire enterprise.

The problem for the application designer is that there is always one more user for the application. Workgroup applications may grow with the addition of more people to the project or the merging of two groups. Intra-enterprise applications may face additional users due to a merger, the connection of business partners, or the opening of the enterprise to the Internet.

Is it possible to place a limit on how many users an application needs to scale to? In practice, limits exist for every application. They might be based on the software algorithms used or on how fast the computer executes. Successful enterprise software has to be capable of handling the largest numbers of users, from tens of thousands to tens of millions of users today, growing to even more in the future. If the software is not designed with this range of support in mind it is likely to fail at one end or the other.
Approaches to Scaling

Figure 10.2 shows various approaches to scaling and how they group into two major categories. One category is getting more work through a single system, while the other category is dividing the work across multiple systems.

Single-System Scaling

An application can be made more efficient on a single system a multitude of ways. Chapter 11 covers many of them. For single-system scaling we want to discuss the main aspects of work distribution, both offloading to the client application and usage of multiple execution threads within the service.

Offload Processing to Client  One of the first aspects that affect scalability is the work distribution between the client and the server. In general, increasing the percentage of a transaction that can be executed on the client will allow a server to handle more transactions. Figure 10.3 shows the effect that moving work to the client can have, even in the presence of additional overhead. This leads to deciding on a thin client or thick client architecture. Thick clients are assumed to perform some application processing that would be done on a server in a thin client model. This means that the application designed for a thick client architecture will allow the

![Diagram showing general approaches to scaling]
servers to handle more transactions (the right side of Figure 10.3) than an equivalent thin client application.

But enterprise software is never about optimizing just one aspect of the overall system to the exclusion of all other aspects. Thick clients require that more software be deployed on the client system. In a large enterprise environment the additional cost of controlling and managing a software application installed on 20,000 or 200,000 systems may overwhelm the value associated with the application. Also, the technique of splitting the work between the client and the server might be used to increase the number of servers that can be effectively applied to the work.

Today’s main approaches to thin-client computing are browser-based applications and server-based computing (for example, Windows Terminal Server and Citrix WinFrame.) Browser-based applications separate the display of the application from the business logic portion by having the server deliver HTML to the client browser.
for rendering. The advent of ActiveX and Java applet support in the browsers also enables a distribution of additional work to the browser. Server-based computing is in many ways a return to the time-sharing systems of the past. In this case the application actually executes on a server, with the user connected using a generalized client. The main advantage of server-based computing is easier management of a centralized computing environment. Browser-based thin clients have an advantage in terms of their ability to support both a heterogeneous environment—browsers exist for all clients—and by not requiring the installation of application-specific software on the client. These advantages are critical when an application needs to grow beyond the enterprise and reach out into the Internet.

**Multithreading and Multiprocessor Systems** Another scalability technique is the support of multiple execution threads and exploitation of multiprocessor systems. We use the term *thread* as a generic term for executable work. Threads are operating-system-dependent and have different names by operating system; they are called *threads* on Windows NT and Windows 2000, *processes* on Unix systems, and *tasks* on OS/390. Threads improve scaling on single-processor systems by allowing work to be done on one transaction while another transaction is waiting for I/O to complete or for some external event to occur. Threads are required to exploit multiprocessor systems, since what good are multiple-execution engines if you can only keep one busy?

The easiest way to support multiple threads is to create small transactions that can be executed within an environment that exploits multiprocessing. By using small servlets and a Web application server, for example, you can get significant multiprocessor exploitation. However, it is critical that the servlets be designed so that multiple copies can execute without causing them to conflict in accessing resources. Chapter 11, Performance Characteristics, covers the types of problems that can arise in multithreaded programs.

**Algorithm Choice** Another program characteristic that can prevent single-system scaling is an algorithm that cannot handle the largest number of data elements. The work associated with most algorithms can be represented as a mathematical formula. For example, the bubble sort algorithm takes the order of $n^2$ operations to sort $n$ elements. The heap sort algorithm requires $n \cdot (\log_2 n)$ operations. Between the two algorithms, heap sort is a better choice for sorting even a moderate number of elements. Table 10.1 shows the time required to run bubble, insert, and heap sorts against different numbers of elements. (The code used to generate the table is included on the CD.)

But how does an application designer choose between two algorithms that require the same order of operations? This requires a deeper understanding of how the algorithm actually works. Each algorithm has both a fixed setup cost and a variable multiplier. For example, one algorithm might take 100 seconds of setup and 0.1 seconds as a multiplier and another algorithm might have 1 second of setup and 0.2 seconds as
its multiplier. As is evident from Figure 10.4, the first algorithm is better for a large number of elements, while the second algorithm is better for a small number of elements, with the crossover point being 1,000 elements. Software that needs to work well at both ends of the usage spectrum may need to choose which algorithm to use based on the number of elements being processed.

<table>
<thead>
<tr>
<th>Number of elements</th>
<th>Bubble</th>
<th>Insert</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5,000</td>
<td>350</td>
<td>130</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>1,763</td>
<td>671</td>
<td>20</td>
</tr>
<tr>
<td>50,000</td>
<td>48,800</td>
<td>20,279</td>
<td>120</td>
</tr>
<tr>
<td>100,000</td>
<td>21,6362</td>
<td>91,822</td>
<td>270</td>
</tr>
</tbody>
</table>

**Figure J.4** Algorithm comparison
Multiple-System Scaling

The other major category of scaling technique is to increase the number of systems involved in executing the application workload. This is a generalization of offloading processing to the client as well as multiprocessor support. Let us examine two general techniques for increasing the number of systems. The first is to use some form of replication of the server. This would consist of installing a second or third copy of the application and routing work requests across the copies. In theory, two servers can handle twice the workload of one. In practice, there is often some degradation due to the need to coordinate data between the servers. Commercial methods of adding additional servers include IBM SP2 processor line, NetDispatcher and Microsoft’s Cluster Service. As discussed in Chapter 9, this clustering can also greatly increase the overall availability of an application, in addition to raising the number of users it can support.

Another way to increase the servers for an application is to build them in an n-tier architecture. Offloading processing to the client is an implementation of a two-tier processing architecture. More common is a three-tier architecture, illustrated in Figure 10.5. With this architecture the application processing is split between client, a middle-tier application server, and the backend database and core transaction processing systems.

Most enterprises will gain horizontal scaling by combining both techniques. First they can divide the application into multiple processing tiers, then spread the processing of each tier across multiple systems. This provides the greatest amount of flexibility by isolating system elements from each other. Additionally, the replicated servers can be located at multiple sites to provide for more responsive system access to various locations around the world.

![Figure 10.5 Three-tier architecture](image-url)
Directories and Scalability

Directories can be used to improve the scalability of an application by acting as a data repository for read-mostly data. This allows searching and management of the data objects to be moved from the application server to a separate directory server. This directory service can be partitioned or replicated across multiple servers, allowing for increased scalability. For example, by partitioning intranet data from Internet data and creating multiple replicas of the Internet data, Internet applications can achieve the directory response they need without requiring intranet data to be needlessly replicated. (See Figure 10.6.) Also, the directory service can be optimized for the tasks of searching and retrieving the data.

Also important to scalability is the ability of the directory to provide a single point of control and administration for common data. This allows multiple transactions and services to use the same data elements without requiring the duplication of the data for each service.

Summary

Enterprise software is often required to scale in support of vast numbers of people. While increasing the number of transactions that a server can handle is one important part of scalability, it is also important for major enterprise class applications to be able to scale horizontally by adding additional servers.
Chapter 11

PERFORMANCE

“All processors wait at the same speed.”
—UNKNOWN

What is performance and how does it effect the creation of enterprise software? We explore some the common types of performance measurements and how they are applied. Then we discuss some of the factors that gate application performance. Finally, we apply these lessons by looking at specific performance problems associated with directories.

Performance and Performance Measurement

It is important to separate the concepts of system performance from application performance. System performance is related to the underlying operating system. It is a measure of the throughput and responsiveness of the operating system and associated hardware. Application performance is associated with the application as seen by users. It is possible for system performance to be good even when application performance is unacceptably slow.

Performance is normally measured as either throughput or response time. Throughput is the measurement of the amount of work the system or application is capable of performing in a period of time. Response time is the measurement of how long it takes to process one transaction.

Throughput is normally measured in terms of transactions per second, or transaction rate. Throughput is normally benchmarked by configuring a very powerful system and then running as many test transactions as possible. For a directory, the critical transaction types are reads and searches. Write transactions are not nearly as critical because they occur much less frequently. One place where it is important to understand the write throughput of a directory is bulk-load scenarios. Bulk-load scenarios can occur both in initial setup and in recovery situations.
Response time is measured as the elapsed time it takes to process a single transaction, preferably as seen by the system user. Response time is generally achieved by focusing on optimizing the processing for a single transaction, sometimes to the detriment of transaction rate.

Capacity is the number of system users that can be supported, given a transaction rate and response time goal. Capacity determines when a system has to be upgraded or replaced in order to accommodate additional workload. Capacity is closely related to scalability, as described in Chapter 10.

**Gating Factors**

Application performance is often gated by one of the following three factors.

1. Efficient processor usage and constraints (multiprocessing, threading)
2. I/O utilization and constraints
3. Network bandwidth, latency, and timing fluctuations

**Processor Usage**

Given that all processors wait at the same speed, can the application generate enough threads (or pieces of work) to keep the processor or processors busy while performing effective work? If processor utilization, I/O usage, and network traffic are all low, then it is likely that more execution threads will allow the application to improve its throughput and response time. Two ways of creating additional work threads are to explicitly program the application to create and use multithreading or to have small transactions\(^1\) that can be scheduled by a transaction system. Multithreaded programs can be hard to create and difficult to debug. This is because multiple threads need to coordinate work such that one thread does not interfere with the processing of another thread. Creating small independent transactions is often easier for developers than coding explicit support, but even these transactions need to be examined for data consistency problems that arise when multiple programs attempt to modify the same data. Multiple threads are a necessity for effective usage of a symmetric multiprocessor (SMP) system. SMP systems have multiple processors, making it possible for two threads to be executing at the same time.

The unique errors in multithreaded programs are associated with incorrect serialization of resources—cases where two or more threads access the same resource and end

---

\(^1\) For this chapter we use the word *transaction* to mean any self-contained piece of work, such as a classic http request processed by a Web server.
up causing improper updates. Sample program serialization error shows what can happen when a variable is improperly accessed. The core of serialization error is

```c
for (j=0; j<NUMLOOPS; j++) {
    temp = counter;
    counter = temp + 1;
}
```

By default, 30 threads each execute the loop for 5,000,000 iterations. If everything goes correctly the final counter value will be 150,000,000. Some actual results are shown in Figure 11.1.

Figure 11.2 shows one possible execution sequence of two threads of serialization error. In this sequence, the count gains one, instead of two, as it should.

A second type of error is that a program just doesn’t understand that a value might be independently changing. Consider the following code fragments from different threads:

```c
for (i=0; TRUE; i++);
```

and

```c
do printf(“%d “,i); until FALSE;
```

While one would expect i to increase, some compilers will determine that i is not changed inside the loop and optimize it into a register that does not change, with erroneous results. The fix in this case is to change the definition of i to include the volatile specifier:

```c
volatile int i;
```

The expected value is 150000000.
On try 1 the final value is 103308973.
On try 2 the final value is 99099401.
On try 3 the final value is 103276592.
On try 4 the final value is 95019044.
On try 5 the final value is 102781607.
On try 6 the final value is 100058884.
On try 7 the final value is 99607499.
On try 8 the final value is 102909740.
On try 9 the final value is 95280758.
On try 10 the final value is 103357755.

Figure 11.1 Output from execution serialization error
For more details on threading errors, see the write-up available in this chapter's section on the companion CD.

Why does a directory user need to understand multiple threading errors? Directory using applications will share data with either other applications or other copies of themselves. This data sharing has many of the same characteristics as multiple threads sharing memory or local files. It is critical for applications that update the directory to understand the serialization problems related to multiple update access of a shared resource. Meanwhile it is also important for applications that only read the directory to understand that the values they retrieve may no longer be current, and to have a strategy for determining when and if they need to refresh the local values.

**Input/Output Usage and Constraints**

A second area that can affect performance is input/output (I/O) usage and constraints. Consider a program that has to read 16 million bytes of data from a device. If the device can handle 10 million bits per second, and the program does not have to do any processing of the data, it will still take about 13 seconds just to bring the data into memory. Another way to improve the throughput and responsiveness is to exploit I/O overlap. If the data is split across two I/O channels, the time to read can be cut in half. The LDAP API and protocol supports asynchronous operations that enable programs to improve performance by overlapping directory operations with
other I/O or processing. Program `ldapasc` (see Figure 11.3) demonstrates the advantage of this overlap by showing LDAP operations being performed at the same time as disk I/O.

The `ldapasc` program has a shorter response time than the equivalent synchronous program, `ldapsyn`. This is because the disk I/O can occur at the same time as the network I/O and server work time, instead of having to occur in sequence. The `ldap_results` function is used to wait for and receive the results of the search.

Network Bandwidth, Latency, and Timing Fluctuations

A third area that can affect system and application performance is network usage. One point of view is that network I/O can be treated the same as regular I/O. However, the lower bandwidth, higher latency, and increased timing variation that is common for network I/O when compared to regular I/O gives network communication a greater impact on an application. At its most basic, bandwidth is the measure of the amount of data that can be delivered in a period of time. If the application has to operate in an environment where bandwidth is limited, then a prime consideration can be to reduce the amount of data that is transported, for example, by performing more processing at a server instead of sending the data to the client.

---

2. If the directory service resides on the computer where `ldapasc` is being executed, `ldapasc` is likely to take longer than `ldapsyn`. This is because `ldapasc` and the directory are both contending for disk access at the same time and thereby impacting each other’s performance.
Network latency becomes an integrated part of an application’s response time. If an application spends a second waiting for a network response, that is a second added to the response time. There is little that applications can do to address network latency: the choices are between eliminating the network I/O, getting as much work done in a single request, and employing a faster network. Eliminating network I/O can often be done by requesting data in advance so that the latency does not affect client transactions, such as caching previous directory requests.

A last network performance issue is timing variation. Many network protocols, including TCP/IP, allow for a large variation in the amount of time between the receptions of transmissions. This tends to make application response time erratic. Even if the application achieves 0.5-second response, users will notice if nine transactions take 0.25 seconds and the tenth takes 2.75 seconds! Timing variations can only be addressed by I/O overlap and reducing the dependence on the network.

**Poor Performance**

The performance issues associated with directory implementations are basically the same as for other applications. The main performance issues in directories are scaling with the number of clients, scaling with the number of elements, query time, and search time. While directory write operations can be performance sensitive, they are normally of lower priority.

Performance of the directory server with multiple clients is generally achieved by the internal usage of multiple threads. A simple design is to create a thread for each bind received. A more complex design is to use a pool of threads, where a thread is assigned to process an active request. In both cases, multiple threads are used to allow the overlapped processing of requests from multiple users.

I/O issues are similar to database systems. For reading and writing, the critical factor is to organize the data to allow for highly optimized query and searching. In many cases, hardware RAID can be used to spread the database across multiple physical disk drives. This can greatly increase the I/O throughput and capacity. However, it is possible to still have I/O limitations. For example, some directories use the database technique of writing a log file containing a copy of all write operations. The I/O rate of the log file will limit the number of write operations per second. It is critical that the directory be able to take full advantage of the I/O capability of the underlying system.

Because a directory is like a database in many aspects, the key database performance technique of using data indices is also used. Indexing is especially useful for searches. The process of using an index allows for a quick lookup. Program `ldapindexperf` does the following two directory searches against an Active Directory server.
(objectclass=user)
(objectcategory=user)

The result is 0 seconds for the objectcategory search versus 11 seconds for the objectclass search. (In Active Directory, objectcategory is indexed, and objectclass is not indexed.) These results show the dramatic improvement in search time when data is indexed. However, the trade-off is that updating the indexed data requires more resources. Most directories allow for specifying which attributes are indexed.

Client-side caching is used to improve the performance of client applications. With caching, frequently used data is kept at the client, eliminating the requirement for the client to actually get the data from the directory server. This technique has the disadvantage of requiring coordination to prevent the client from using stale or out-of-date data. It is also possible for the directory server to cache the results of frequent and expensive operations. This caching does not reduce network I/O but can improve the efficiency and response time of server actions.

A general technique used by directories to reduce the effects of network bandwidth, latency, and timing variations on application performance is to locate directory servers near the using application. Chapter 19 on directory replication and partitioning covers this in detail. The goal is to locate servers where they have high bandwidth, low latency, and minimal variations in network communications to the most frequent clients. The complication is that clients must be configured to locate and use the closest server.

It is important for a general-purpose directory to allow for tuning or to do self-tuning. The installation needs sufficient data to determine the attributes that require indices. Also, request information can be used to locate servers close to the most frequent data users. Without the ability to tune the directory server, the only performance management tool is to increase the directory server hardware capacity.

Summary

This chapter reviews the performance challenges associated with enterprise directories. It outlines the key techniques of multithreading and I/O overlap that are used as solutions to various performance issues. General-purpose directories must be capable of providing high levels of performance and therefore are developed using these key techniques. Lastly, it identifies how network placement and replication are used to address performance issues that occur during deployment.
Enterprise system requirements come from the use of applications in critical business processes. As business processes move to on-line systems and are exposed to customers and business partners, the risks increase. On-line data is subject to confidentiality, integrity, and privacy threats beyond those that apply to off-line paper files. The increasing use of the Internet to connect customers provides attack avenues that did not exist when the data was confined to the internal corporate network. This combination of increasing threat and more important data brings with it additional security requirements. These requirements have a direct bearing on enterprise directory usage.

Policy and Risks

An enterprise’s security goals need to be defined in an overall security policy. The major goal of the policy should be to define how the enterprise approaches the issues of protecting its data assets. A security policy needs to address these main areas.

- Data confidentiality—preventing unauthorized disclosure of the data
- Data integrity—preventing unauthorized modification or deletion of the data
- Accountability—being able to identify and hold accountable the person who modified or disclosed data
- Privacy—keeping an individual’s data private and preventing it from being used for unapproved purposes (closely related to data confidentiality)

The lack of a policy leads to both inefficient and ineffective data protection, inefficient because some low-risk data might have expensive protections applied—perhaps the employee phone book data will be greatly overprotected, wasting money and resulting in lost productivity—and ineffective because other information might be under-protected—customer data might be exposed on an Internet Web site. Equally important to having a security policy is having executive management support for the
policy. Implementing the policy will incur expense and inconvenience, and when this occurs, it is critical that executive management supports the security requirements.

A good policy will apply to the data regardless of the data's form. If the data should be kept confidential, the policy should state that it should be confidential, whether it is stored in electronic form or in hardcopy. Once an overall policy is in place, a risk analysis can be done. The risk analysis would cover the types of threats, the likelihood of the threat occurring, and the business loss or cost associated with the threat. Often only a high-level risk assessment is required. Often it is not crucial if the potential loss would be $10 million or $50 million, since either value would be a significant loss, worth preventing.

Some of the threats that information in a directory can be exposed to, illustrated in Figure 12.1, are:

- Data confidentiality—intercepted during transmission, access by an unauthorized person due to incorrect access control settings, improper access by directly accessing the directory’s backend data store
- Data integrity—modification during transmission, improper modification allowed due to incorrect access control settings, improper modification by directly accessing the directory’s backend data store
- Accountability—lack of audit records makes it impossible to identify who modified data; stolen passwords makes it impossible to determine who really accessed some data
- Privacy—usage of data for unapproved purposes, improper disclosure of private information

![Figure 12.1](image-url)  
Selected threats to data stored in a directory
Most risks have some set of countermeasures that can be used either to eliminate or at least to reduce the risk. Sometimes the threat can be reduced to an acceptable level of risk. For example, most companies find it acceptable for employees to have confidential data in an unlocked office during the day. The key countermeasures for risks associated with the four policy areas are

- **Data confidentiality risks**—data encryption, system and application access control facilities, database views
- **Data integrity**—cryptographic hashing and signatures, system and application access control facilities, database referential integrity mechanisms
- **Accountability**—auditing logging or recording, strong user identification and authentication
- **Privacy**—controlled usage of data, system and application access control facilities, data encryption

**Application Development and Maintenance as a Source of Ongoing Risk**

Application development and maintenance introduces multiple sources of risk, some internal and others external. Intentional internal attacks tend to be associated with software developers adding code to perform unauthorized functions. However, it is even more likely that programming errors will create an unintended weakness in the software application. External attacks tend to be either untargeted, such as a computer virus, or targeted to cause a specific program to operate incorrectly, as in most buffer overflow attacks. External attacks are unlikely to be accidental.

Inside development attacks use the developer’s job responsibilities to add code either during the original program development or during ongoing maintenance. This risk is generally managed by ensuring that all program changes are peer-reviewed before being placed into a production or secured environment. Additionally, peer reviewing is also an effective tool for the generation of quality software.

However, more security risks are introduced by poor software design or coding than by a developer’s intentional actions. Many software design practices can lead to increased risks. For example, designing a program so that it requires users to have access to data outside their job responsibilities adds additional confidentiality risks. Requiring write access to data can increase the chance of integrity attacks. In many cases, a strong security policy can assist application designers with reducing these risks by identifying how data should be managed.
With the increased exposure of applications to external usage, coding errors become more important. If no one can gain the access required to attack a program, then the error is unlikely to be exploited. However, when an application is made available on the Internet, the number of potential attacks is greatly increased. These attackers can either be intent on specific damage or just interested in any damage they can cause. In many cases they do not even view what they are doing as causing damage.

Common countermeasures for coding errors are

- Defensive coding—understanding the types of code-level attacks and how to write code that is not open to them
- Code quality—careful attention to detail, including reviews
- Code or application isolation—ensuring that code with potential weaknesses is isolated from attack, including placing critical applications behind firewalls

For externally written applications, an additional countermeasure is the vendor reputation for writing secure code.

**Controlling Access Based on Organization Responsibilities**

A key part of implementing security is identifying the users that are processing data. It is hard to limit data access to only authorized people, if you cannot reliably identify who is attempting to access the data. Today most systems authenticate users by comparing a password value to a stored copy of the password. Some systems use a centralized authentication service to avoid having to define users at each individual server. The proliferation of Web servers has increased the need for centralized authentication. Four main ways of authenticating Web users, as depicted in Figure 12.2 are as follows.

1. Authenticate them using a security database on the Web server.
2. Pass the authentication off to an existing backend system or database.
3. Use a dedicated authentication service such as Radius.
4. Define the users in a directory and use directory protocols for authentication.

Each method has advantages and tradeoffs. Using a local security database provides speed but requires duplicate administration for multiple servers and exposes the

---

1. Most systems use multiple methods to allow the passwords to be stored in a nonreversible fashion. However, they still require a copy of the nonreversible password locally.
authentication database to attacks on the Web server. Accessing existing back-end systems has the advantage of allowing integration with existing security procedures. While this is good for internal users, companies might not want to add millions of customers to their existing system security systems. The addition of a dedicated authentication service allows the new Web users to be defined separately from existing internal users. However, additional information still has to be defined about the users. Using a directory has the advantage of both isolating users and enabling the storage of additional attributes about the user.

One easy way to use a directory for authentication is for a Web-based application to collect the user’s authentication information (user ID and password), then use the LDAP bind operation to present them to the directory for authentication. This allows the application to verify the user’s identity without requiring a local definition of the user. The password needs to be protected from network interception using technologies like SSL. Directories are beginning to support stronger authentication methods. For example, Active Directory supports Kerberos, and the IBM SecureWay Directory supports client public key certificates. Additionally, the application can then access other attributes of the user to either personalize application usage or to provide additional application specific access control.

The directory can also be used as a central access control mechanism. If the application needs to control access to a resource, it can attempt to access a companion object
in the directory and use the resulting access success or failure as the access control
decision. So to grant use access to a resource, the administrator grants access to a sur-
rogate in the directory. This method requires the directory access control model to
be able to specify the desired resource protections. It is best suited for cases where
the protection granularity is not required to be very fine grained. For example, the
technique is good for specifying read and write protection, but not as good for con-
trolling 20 different flavors of write access. Appendix F describes the access control
features of several existing directories.

Program `ldapauth` (on the CD), illustrated in Figure 12.3, provides an example of
how all three techniques can be combined. First it uses the following code to bind to
the directory with the user’s ID and password.

```c
rc = ldap_simple_bind_s(hLDAP, wcUserDN, wcPassword);
if (rc!=LDAP_SUCCESS) { ... }
User has been authenticated
```

Once the user is authenticated, his or her telephone number is retrieved. The phone
number is used to personalize the access check. The access check determines the
user’s access based on his or her ability to access an organizational unit, named using
their area code. (The directory objects are named `OU=areacode,OU=AWLEnterprise-
Directories CH12`.) If the directory allows access to the directory object, then the
secret message is printed.

![Diagram](image.png)

**Figure L.3** Security actions implemented in `ldapauth`
The Audit

An important component of implementing an accountability policy is to audit the data accesses that occur. Auditing can be used to achieve three goals.

Control—to document the proper working of a control method
Detection—to detect a violation of policy
Repair—to record enough information to allow a security violation to be repaired

Figure 12.4 shows how a single audit record can sometimes be used for all three audit purposes.

Most computer system audit records are created for control purposes. In these cases, the audit records serve mainly to document that the access controls and other security measures are working. (Namely that inappropriate accesses are being denied and legitimate requests are allowed.) Control records can sometimes be used to detect that someone is attempting to violate the security policy, for example, by recording access attempts to files and systems. The key thing to remember about control-style audit logs is that they document decisions; they do not document that legitimate authority was misused, nor do they document access that bypassed the control point.

Some auditing is done for actual detection. Recent times have seen the creation of a market for network intrusion detection tools. Whereas control-style auditing documents decisions, detection auditing tends to record activities, then to extract trends from that data. For example, an increase in late night traffic to an obscure ftp server...
might mean that it has been taken over by a hacker. The biggest disadvantage of
detection auditing is determining what to do after you detect the policy violation. It
is nice to know what has happened, but wouldn’t it have been better to have pre-
vented it to begin with? The critical point is that a balance needs to be struck, espe-
cially since the two techniques complement each other. The instigation of control
methods can prevent most damage, and the detection technology can help identify
some of the remaining cases.

Which brings us to the third reason for creating audit records, having sufficient data
to be able to recover from a security breach. In general, the IT industry does very lit-
tle recovery-related auditing. While there is a significant effort on reliability and
disaster recovery, it is not from an auditing focus.

What types of events does auditable software need to record? Unfortunately, the only
valid general answer is “It depends.” Basically, every security-relevant event needs to
be recorded for later review. Each time the program makes a decision about allowing
access or modification of an important piece of data, the decision should be audited.
In deciding what data to record it helps the application developer to think of which
purpose (control, detection, or repair) the record will be used for. The 1dapauth
sample also records a Windows audit record (to the application event log) for each
access it grants or denies. This record can either supplement or replace any auditing
performed by the directory.

Summary

Enterprise security requires the identification of users, controlling their access to
resources and data, and auditing their actions for detection, prevention, and repair of
unauthorized actions. Directories provide a powerful tool that can centralize user
authentication and access control, while providing additional capabilities to applica-
tion designers. The desired security characteristics will need to influence the direc-
tory implementation chosen as the enterprise directory. If the directory is going to be
used as a security control point, it will need to be secured and managed as part of the
enterprise’s security infrastructure.
Chapter 13  
**LIFE CYCLE CONTROL**

Today's enterprise applications and their distributed operating environments are so essential, numerous, and complex that additional infrastructure is needed to manage them. A major problem in many organizations is the management of applications. How do we track what applications are where and at what service level? Where do we get early detection of when they're causing a problem? Is it possible to continue to service client requests even if there is a malfunction? How do we know the nature of the problem? We need a place to make marks on a common wall. Cue the directory. A measure of human common sense doesn't hurt either. The directory itself is a complex application after all. The term *software life cycle* describes the software cradle-to-grave experience. It encompasses software development, installation, configuration, management, and maintenance. This chapter takes us through the management problem of software life cycle control. The case study in Chapter 26 covers the related problem of application management.

**Serviceability and Maintainability**

Two aspects of the software life cycle are interesting. One specific aspect is how a directory service itself is established and maintained. The other, more general, aspect is how applications that use the directory service are installed and maintained. The former task is actually a specific example of the latter problem because the directory itself is a distributed application.

Any piece of vital software, such as a directory service, needs to support robust usage scenarios. Pieces of serious enterprise software may have been written that fail to take life cycle issues into account. Installing a piece of software—a directory server for example—is probably the last inexpensive activity in the life of that application. Once installed, it may need upgrading to a new release. It could fail occasionally, which would require debugging and the application of fixes. A failure could possibly require public relations damage control or legal help if a critical application suffered.
How many times do you want to experience a software problem before finding and applying a fix for it? Inexperienced software engineers think this question is of little importance. “Just reboot at least once a week, and it will be fine. If not, get me a core image the next time it crashes.” An inexperienced software engineer may be an expensive liability. Minor malfunctions in hardware or software result in annoyance, ill will, or outright lost commerce. Downtime is unthinkable for some applications. Remember our discussion of man-rated software in Chapter 5 and our assertion in Chapter 6 that a directory service could elevate the usage rating of software? We’re not even thinking of the Space Shuttle here. Let us simply consider high availability for large-scale enterprise and commerce applications for now. Consider the negative publicity and hysteria surrounding even short downtimes for several on-line trading ventures.

**Controlling Downtime**

A good-quality distributed application is directory-aware or directory-enabled such that it leverages the services of a good directory. This may not be an obvious conclusion at first. Let us walk through a set of increasingly severe failure scenarios. As we proceed, watch for possible single points of failure. We want to attack these by making them redundant. The goal is to isolate failures to as few client requests as possible while maintaining service to the greater population. Part of the task is early detection of failures. It is not unusual that a failure may be undetected until it cascades to a larger, obvious, severe, and expensive failure. We want to nip that problem in the bud. Our strategy is to capture failure data coming from anywhere and then to be able find it, extract it, and analyze it from anywhere. The name of the game is logging and analysis.

**Real-Time Event Logging with First Failure Data Capture**

A useful directory will have object classes in its schema or allow an extension to the schema such that any directory-enabled application may leverage it for event logging. We’re interested in using a directory to aid in the capture of interesting events and failure information. Remember that the directory is not a structured database. It holds read-mostly data that may act as a locator for more volatile subordinate data, such as a published event logging service. (See Figure 13.1.)

A distributed application binds to some directory replica. Preferably, it chooses the nearest directory by using some network response-time criterion or DNS topology. Remember that the DNS is a type of directory. In some cases we may use it as a coarse-grained top level of the Active Directory.
Next, the application locates an event log service through the directory. Now it simply logs interesting events, exceptions, and failures to the logging service. (See Figure 13.2.)

Other query or report-generator applications locate the event service the same way. A remote administrator may run queries or reports, filtering the output by application,
location, time frame, or a combination of criteria. A really great application could ship a daemon service that scours the log for problems and then sends an e-mail or a pager message, or otherwise notifies the administrator to look at the log. More likely, an enterprise administrator could script this type of action. Perl, Python, or Tcl are good tools for this on Unix-based systems. JavaScript or VBScript could be used to script COM API wrappers on Windows NT or Windows 2000 servers. A red flag caused by excessive retries, errors, or response-time degradation may be detected, perhaps early. Without this infrastructure, problems could endure until front-office telephones ring, productivity declines, and revenue drops, or the media publishes an unfavorable article. Of course some humans must play their part by pulling the information from the event server and carrying out reasonable actions.

Failing a Request Versus Failing the Server Application

What should happen when a client request fails in the server? Maybe the server should log the error and exit? No, of course not. That makes it easy on the developer, but the coding life isn’t always easy. Logging an exception and then failing the server application makes the log shorter and the report short and easy-to-read, but this cuts service availability.

The server should be coded to fail the request, log the error or exception, and attempt to recover normal operation. Often servers are structured as a boss thread or process that listens for a request and forwards it to a pool thread or process request-handler. The request-handler will suffer most of the errors or exceptions in a typical server. It could log the error and die, provided that the server’s pool-manager provides a replacement in the pool. The application thus has a chance to continue to service other client requests when one client request fails.

If the event logging process or thread fails it should be restarted. The replacement should carry out a logging operation that logs the failure of the event log request itself.

So far, we may fail and replace worker threads and fail and replace the event log requestor. At this point the boss thread or process is the weaker link. The server design should eliminate single points of failure as much as possible. A means to restart or replace the boss should be included. By now we see that the failure must be logged first.

Now we may determine that the pool-manager is the next-weaker failure point. Its job, presumably, is to expand and contract the worker pool according to some heuristic. Perhaps its failure mode kicks in when the system runs low on resources. This may be difficult to program around since the problem is global to the server application. It is safe to assume that this type of error is far less frequent than one that would
cause a single request to fail. The server may have no option but to abort. First, it must log a critical error to the logging service that it located through the directory during initialization. This error should contain as much information as possible, since this is a more important event than that of merely failing a request.

**Auto-Restart** Once our server has logged a failure and aborted, what’s next? Remember that the goal is to keep our server in service. How our server was started in the first place depends on the host operating system and the administrator that maintains it. A system initialization script or a service monitor may be configured to start servers automatically when the system starts.

When our running server logs a failure and aborts, a good first tactic is to try to restart it a number of times. These attempts may be logged. Each attempt should automatically be carried out at intervals to give a transient problem stimulus a chance to diminish. But what carries out this automatic restart? The starting mechanism used at system startup possibly doesn’t know that the server has aborted. Or does it? A subsidiary script could wait for the server to end, inspect the ending status, and conditionally restart the server. There is an alternate tactic.

Some platforms have a task or job scheduler that schedules programs to run at a future time. Windows 2000 and Windows NT have the at command. Unix systems, including AIX and Linux, have the at and chron facilities. Why not have the failing server schedule itself to restart at a future time as a part of its abort processing as shown in Code Example 13.1?

What if repeated attempts to relight server flameout fail? When a problem is so severe that the server cannot restart requests, it aborts itself and it attempts to restart a certain number of times. But perhaps we’ve encountered a hardware or resource problem that prevents the server from running at all. We’ve tried two increasingly drastic lines of defense against the problem. Usually the first or second tactic will work. Client users may not know there was a problem. But if we have a problem that causes us to certify the server as dead, what tactic should we use?

**Automatic Fail-Over** Our server may be a member of a large enterprise or a server farm. It makes sense to try to switch the workload of our server over to another server. From the client point of view, the service is still operational. Response time may be larger or temporally erratic. A user may notice a small delay as workload is redirected to other servers.

How is the switchover carried out? It could be done on a per-service process basis or on a per-computer basis. First let us discuss the switching of all workload among computer hardware. This fail-over clustering is one function of hardware clustering. The increasing popularity of clustering is based on math that says several cheaper, slower systems that appear as one system are more reliable than one large, expensive system.
Some operating systems have built-in fail-over clustering as mentioned in Chapter 6. Multiple computers appear to the user as one and share a rack of disk drives on a SCSI bus. This may appear to be a single point of failure. But the disk drives should not be a weak point because they use a redundant RAID configuration. One drive can disappear without a loss of data or a loss of service.

How does the cluster detect that a member machine went down? A heartbeat “I’m alive” pulse flows between the computers on an RS-232 cable or an Ethernet segment. When a heartbeat disappears, the failing computer workload fails-over to the other computers in the cluster sharing the disk drives. (See Figure 13.3.) The entire cluster runs slower but stays operational. An Ethernet hub for the heartbeat signal is a single point of failure. A crossover cable could be used instead of a hub for the heartbeat, or a more expensive redundant fail-over hub could handle the heartbeat signal.

```c
hEvent = CreateEvent(NULL, true, false, NULL);
beginthread(haltThread,0, hEvent);
for (int restart = 0; restart < 5; restart++) {
    // Start service, passing parameters
    STARTUPINFO si;
    memset(&si, 0, sizeof si);
    si.cb = sizeof si;
    PROCESS_INFORMATION pi;
    memset(&pi, 0, sizeof pi);
    BOOL rc = CreateProcess(
        NULL, pzCmd, NULL, NULL, true,
        NORMAL_PRIORITY_CLASS, NULL, NULL, &si, &pi);
    // Wait for service to end
    if (pi.hProcess) {
        HANDLE harray[] = {pi.hProcess, hEvent};
        DWORD ec = WaitForMultipleObjects(
            sizeof harray / sizeof *harray,
            harray, false, INFINITE);
        if (ec == WAIT_OBJECT_0) {
            TerminateProcess(pi.hProcess, 4);
            break;
        }
    }
    // Sleep for restart delay read earlier via LDAP
    Sleep(rstdel);
}
CloseHandle (hEvent);
```

**Code Example M.1  Auto-restart**
What if our server isn’t aborting because of problems catastrophic enough to bring down the entire computer? How do we gracefully switch the workload? Each client should locate the server through a directory query when it makes a request. If the failed server is part of a farm, the client will find another server like it in the response.

**Directory Redundancy**

We’ve maximized the availability of a service by using restart retries and resorting to fail-over if those tactics fail. We used a good directory to maintain the state information. Isn’t the directory a single point of failure? The directory is a critical service. It cannot stand as a single point of failure. We need a good directory constructed as if it were man-rated, as discussed in Chapter 5. We need two or more good directories to mitigate the likelihood of a directory machine failure bringing down an application or all applications.

The directories should be configured to carry out multimaster replication, both for redundancy and to distribute directory replicas closer to requestors. (See Figure 13.4.) Replication and partitioning are covered in Chapter 19.
Periodic Maintenance

A collection of enterprise applications needs maintenance like a piece of machinery does. An enterprise IT operation is an enormous finite-state machine, but it may appear to have the frailties of a living thing: Performance problems, data problems, and malfunctions may appear to come on like a flu epidemic. How does a directory aid the maintenance task?

Backups and Directories

The data used by a service may be a source of problems or a victim of problems. The data store is a failure point. The data often resides in a database serviced by a DBMS. It should be backed up in a fashion that may be easily located and reliably restored. There should be no question as to which backup volume was created last, which volume was created before that one and so on. Date, time, description, and locator information should be written into the directory. A backup operation is a relatively infrequent event compared to a nominal replication interval.

Multiple generations of backups should be maintained since problem-causing data could be backed up along with good data. An older backup may be used to restore the application to a good, albeit down-level, state.
To back up the directory itself we allow multimaster directory replication to create redundant directory data in other locations on other machines. What about accidents happening to the directory data? We don’t want to spread bad data throughout an enterprise. We would use utilities to back up all or parts of the directory information tree.

A common LDAP directory import/export file format is LDAP Data Interchange Format (LDIF).¹ There are tools that ship with directory services that import and export LDIF files. How could we use an LDIF utility to our advantage to relieve the grip a bad directory entry would have on the health of our enterprise?

If an administrator deletes a departed contractor’s account instead of disabling it, the account’s unique security ID is lost. If the administrator recreates the user account, a distinct new security ID is created. The old user’s file ACLs would not necessarily allow access by the new security ID.

A manager now wants to place another contractor the departed contractor’s job. The account must behave as before. The manager needs the old account restored to the directory. If the directory had an LDIF backup extracted periodically, an administrator in authority could restore only the departed user’s account from the LDIF backup. This should be carried out carefully. The authority should be given to somebody with experience and with controls in place. LDIF backup information should be logged to a directory so that it may be found and identified at any time from anywhere.

**Planning for the Next Release**

We’ve seen how a replicated directory may contain the state of distributed applications to maximize early failure detection and minimize single points of failure. Next, we’d like to fix software problems or add functionality by installing a new release of our hypothetical server.

**Carrying Out an Upgrade**

If a server is directory-enabled, its settings and state are maintained in the directory. (See Chapter 26 for details about application management through a replicated directory.) Assume the server’s operational parameters, its event and exception log, the location of needed files or database servers, and the location of its own binaries are found through a search of a directory as shown in Figure 13.5. To upgrade the server is to replace its executables or to install an alternate version of the server

---

somewhere else and activate it in place of the old server. The latter is a good strategy because the switchover may be instantly reversed if there is a problem.

**Installing a New Server in Place**

To upgrade the binaries of our directory-enabled server we use an administrative sequence of stopping the server, upgrading the binaries, and finally restarting it. The server will obtain some of its operational state from the directory. Not all of the state is necessarily resident on the directory. Remember that we located the event log through the directory, but didn’t actually log to the directory. One should review update frequency and data size before making a decision to place actual data in the directory instead of a reference to its location. Events need a separate log. Other parts of the server’s state could require a level of indirection. Operational parameters such as file or database paths would go into a schema attribute designed for that purpose.

The client part of the application should also be directory-enabled. Simply stated, this means it always searches a directory to locate its server. An old release of the server could be on off-line standby in case of a problem in the new release. The client’s search would return the location of the new server. The administrator could make either server active by carrying out an LDAP update. The client’s queries are cached by the LDAP service. A repetitive search doesn’t impose a significant performance penalty.
We have not described how the server location is actually changed in the directory. How does an administrator carry this out?

**Streaming Data to a File**

We may export part of the directory to an LDIF file. This file is flat ASCII or UNICODE text. The administrator may export the subtree with the server operational information. She could edit the file with any text editor to alter the location of server binaries. She would import the updated LDIF file into the directory. A separate prior LDIF export is sensible backup insurance. (See Figure 13.6.)

An improved solution is an LDAP-based GUI utility that carries out the directory update. The utility is part of directory-enabling the distributed application. Validation handlers in the GUI would reduce mistakes and log its activity for an audit trail.

**Applying the Life Cycle Lessons**

We've seen how to raise the level of the server to enterprise-level quality and availability, but the server is only half of the application package. Our client is also directory-enabled, or it would never find its server, or never switch server instances when a server goes down or is replaced.
Client and Server Changes

We replaced part of the distributed application when we upgraded the server. This is acceptable, provided that a server upgrade doesn’t provide new functionality or that it provides no new function that breaks backward-compatibility with clients. The application should be designed so that a new server function doesn’t break down-level clients. Those clients simply don’t participate in the new functionality. That way, the upgrade may proceed in a step-wise fashion: Upgrade the servers, check the logs to verify correct operation, and only then update the clients.

We want available, bulletproof applications without increasing the total cost of ownership. How do we update the clients while maintaining availability and responsiveness without a human visit to each client machine? If the clients are HTML and Web-based, the hands-on logistics are simplified. The Web client-server paradigm delivers the client code from the application server whenever it’s used—maybe with some client-side caching involved. If the clients are thick-client applications instead of Web applications, the binaries may reside on a file server. No physical touching of machines should be needed for either configuration.

Let’s consider a Web client first. If part of the Web client logic is scripted, a new HTML page with new script will be loaded. If a Java applet is part of the logic, the updated HTML will specify a new version of the applet if necessary. If client logic is resident in an ActiveX control, it will be cached on the client machine. The updated application’s HTML page that houses the ActiveX control would specify a newer version of the control, when needed. Everything hinges on a version change to the client HTML page. A newer date on that page invalidates its client-side cache image, thus ensuring that the new page is loaded from the server. Or the page could be written with an HTTP header that specifies immediate cache expiration—no caching of the page. Any needed scripts, Java applet logic or ActiveX logic is imbedded in the page, so these entities will be correctly upgraded along with the page. This characteristic is part of the attractiveness of HTML-based clients.

Where does the directory fit into an HTML-based client upgrade? The directory should maintain a record of the software level of all client and server components. It should have server entries that indicate where the HTML pages and components reside. We stipulate that enterprise-level Web-based application clients must consult a directory to locate its published service and required operational parameters. The search may be written in script, a Java applet, or an ActiveX control. We have one application that may consist of client and server parts. These must work together as an application at a published configuration level. Chapter 26 discusses configuration management in more detail.

What about thick clients? The desktop users should access the code from a script or a link to a launcher that references the directory. This script or launcher link is analo-
gous to the HTML page of the Web-based client. The directory search will return the location of the client code and any operating parameters necessary. (See Figure 13.7.)

If this sounds like the operation of the server side of the application that’s because it’s the same idea. We homogeneously use the directory as a point of reference for locations, parameters, state, and failure information of the entire application.

**Availability**

Directory-enabling one application may seem like extra work, but it’s actually inexpensive when the costs, bad publicity, and hard-to-find problems of malfunctioning enterprise-level applications are virtually eliminated. After an enterprise service advertising and management scheme is adopted, directory-enabling subsequent applications may become routine.

**Summary**

Enterprise-level distributed applications that fail or degrade drastically may cause bad publicity, loss of revenue, or loss of work. Single points of failure should be eliminated by redundancy. A good multimaster replicated directory may aid in eliminating single points of failure.

A directory-enabled application will maintain configuration data in the directory for both its client and server sides. This data, along with the fail-safe nature of the client and server part, allows in-place upgrades of either side or both sides of the applica-
tion. Early detection of degradation, early failure detection, and seamless upgrade or fallback activity may be achieved through the use of directory glue.
Directories are working code. They solve problems. They improve data sharing and the ability to perform heterogeneous computing. But they are not computing’s equivalent of a kingdom, or a phylum, or even a class. They exist because a gap between database, transaction processing, and file systems needs to be filled. Directories exist because none of these other data storing and processing systems fit all of these requirements at the same time.

- ACID—only sometimes in special ways and usually as a trade-off against high performance and a high degree of distribution. Read-mostly optimizations are used for the same reason.
- High RASSS
- Define new kinds of data arrangements (through metadata, schema, and directory information trees, or DITs)
- Cross many platforms, leaping over the underlying data problems of ASCII, EBCDIC, Unicode, floating point format, and so forth for a single common understanding of data
- Be multivalued (for example, include several phone numbers for a single user)
- Powerful search and ordering capabilities

What if you had a database that had the ability to be as ACID as necessary for the requirements, while also cross platform, provide very high RASSS, had the functions of directories, and was inexpensive to buy and operate. Would you use it? Of course you would.

As directories, databases, transaction processing systems, and file systems evolve, they each acquire some of the best features of the others. For example, journaling and recoverable file systems can make file access look transactional and some database systems relax their ACID conformance and increase performance for less-than-critical or read-mostly data. Directory vendors are also adding all sorts of features to their directory products.
Predicting the Future

While no one has a perfectly accurate crystal ball, we can predict some features that will be added to directories to make them even more useful than they already are today. Some of the most obvious additions to directory services will be the things that database, transaction processing, and file systems already do well.

Many of these features will be added in the next few years to directory products and will be somewhat ahead of the standardization curve. Some features will be proprietary to one directory service or another, while many features are working their way through the IETF standardization process. Some of the following features will be coming soon, and some are already in some directory products.

Integration with the Operating System

Because directories solve a very pervasive problem of data sharing, they are extremely useful to the running of an operating system. Some operating systems already have embedded directory services; others will have them in the future. Some operating systems will standardize on commonly defined directory services, such as LDAP. Standardizing on common services will allow better interoperating system cooperation and management.

Directories embedded into operating systems improve customer value, even if they don’t work across different systems. For example, a single operating system can use a directory service to distribute software to other instances of the operating system. Ideally the directory would be used to distribute software to many operating systems. (See Figure 14.1.) This software could be in the form of executables that any OS can run, such as Java byte codes.

Transactional “Flavors”

You know that transactional data conforms to the ACID properties (atomicity, consistency, isolation, and durability). Some kinds of data require something less than 100 percent ACID but more than a don’t-care rating. Your home address is a very good example. It’s important that it be correct, but no one is going to die if it isn’t right in some far-flung server at the unfashionable outer edges of the corporation for 15 minutes; it only matters that it be right when someone needs to use it. Say your address only changes once every two years, it is only used by some application once a month, and the change takes 15 minutes to propagate throughout the enterprise. Perhaps the address-modifying program only runs at night and the programs that use the address
only run during business hours. Your address seems to be a good candidate for something between 100 percent ACID and don’t-care.

Flavors of transactional data may appear. Here is a list of possible ways that data can be somewhat ACID.

Category 1—Changes are atomic, but only to one directory object. In addition, the atomicity can be destroyed if two (or more) replicas of the same object are changed on different servers at roughly the same time. This level of atomicity is defined in LDAP version 3. It is up to the owner of the data to use it within these rules.

Category 2—Changes are atomic, but to multiple directory objects on the same server. Again, the atomicity can be destroyed if two or more replicas are modified at roughly the same time.

Category 3—Changes are atomic, but only to one directory object. The atomicity is preserved, even across multiple replicas.

Category 4—Changes are atomic to any number of objects and across all replicas. Clearly this is the hardest to implement and probably the costliest in terms of performance and impact on the directory service’s ability to satisfy scalability and other RASSS characteristics.

As always, some data are simply more important than other data. Instead of having all directory data fit in any one of these categories, we will likely see that some directory attributes are of one transactional flavor (Category 1, for example) while other attributes of the same object are of another flavor (Category 4, for example).
This attribute-by-attribute categorization would allow a pay-as-you-go approach, where important data can be protected to a higher degree than other data and the costs of protecting each kind of data are paid only when required.

Another approach may be on an entry basis (such as a specific user entry), or an object class basis (such as the definition of user entries), instead of attribute-by-attribute. That is, changes to vital object classes are protected in different degrees. For example, the user object class may be more important than the router object class, or vice versa.

**Storing Large Objects**

Directories are typically not optimized for very large objects, such as program executables, high-resolution graphic images, video titles, and so on. This limitation has more to do with old assumptions about the kind of data being shared than about the usefulness of sharing these objects. Directories will be extended for very large objects, either by changing the directory implementations, or by transparently pairing the directory with distributed file systems.

**Dynamic and Easy Partitioning**

Partitioning of data (see Chapter 19) is necessary in many instances for performance or ease-of-management reasons. However, partitioning can be a human-intensive process (that is, expensive). It is not a process that lends itself to iteration: Once a partitioning scheme is set up, it is difficult to go back and change it.

Something probably has to give in this scheme. Partitioning needs to be easier and more dynamic. Exactly how that will be done is anyone’s guess, but one method may be to let the directory service monitor its usage patterns and adjust partitions internally and transparently.

---

1. It bears repeating, since this can be confusing, that the terms *object* and *class* as used here are not exactly the same as you might know from object-oriented programming. An object class is a definition for a kind of entry (or object) in a directory. For example, there can be a directory entry for the Joe user (sometimes called a directory object). In the schema, there is an object class that defines “user” and the Joe user is just one instance of it. See Chapter 16, Schema Fundamentals, for an explanation of how the term object class is usually used in the context of directories.
Hot Spot Management Through Dynamic Replica Creation

Hot spot management through dynamic replica creation may seem as far-fetched as dynamic partitioning, but if you could give a pool of eligible servers to a directory service, the service could monitor its usage and detect hot spots, or directory servers that are overused on a regular basis. Why make a human say, “Oh, I need another replica here?” The directory service could take care of the replica creation, within certain limits defined by the pool of eligible servers. (See Figure 14.2.)

Better Multivendor Interoperability

Directories are little roads: They can be isolated, or joined together. Clearly joined roads have more utility than isolated roads; the same is true for directory data. As more and more directories use standard wire protocols, like LDAP, better interoperability between vendor implementations is a very reasonable requirement.

Figure N.2  Dynamic replica creation
There are several kinds of interoperability. One kind is the simple (and LDAP-defined) mechanism for linking data using referrals. Referrals are a way to put a special object in the directory that says, effectively, “I don’t have your data, but that directory over there does.” Another kind of interoperability is administrative. For example, do I have to bring up two separate tools for administering two directories from two vendors? You probably do, unless you purchase special add-on products.

Unfortunately, even referrals are harder than they sound today. It isn’t as easy as “the data you are looking for is over there.” One reason why it isn’t this easy is that someone had to link the directory servers together; that is, someone had to create the referral object and it may be stale or otherwise out of synchronization. Another reason why referrals are not trivial has to do with security. Different directory servers from different vendors have different security models today. When you, as a client, get a referral back from one directory for another directory, you probably have to authenticate to the second directory as well as to the first, using a different security context (yet another user ID and password, for example).

Simplifying the use of referrals is another area that is ripe for change and improvement. Fortunately, standards are making their way through the IETF process, such as a common access control list (ACL) model that may make programming security on directory objects in separate server implementations easier (such as setting up and using referrals).

Still another kind of interoperability concerns the management of the data. Just because you have two directories doesn’t mean that you have two enterprises; but that is the way some directories work. As an enterprise administrator, you would like to define data that live in different places, regardless of whose directory server stores it. You would like these directory servers to keep data synchronized between them. For example, if you defined a user object, why can’t that user object be replicated across two different vendors’ directory implementations?

There will be improvements in the ability of directory products to share information across vendor implementations. Improvements will be needed in referrals (security, easier administration), general multiproduct administration, and more common schema. Another improvement may be XML-based.

**XML and DSML**

Extensible markup language (XML) is a standard from the World Wide Web Consortium (W3C). XML is a way to package self-describing data. Think of it this way: XML is to data what HTML is to GUI. In fact, extensible style-sheet language (XSL) enables you to display XML by producing HTML. What can go in XML is defined by XML schema elements.
Directory services markup language (DSML) is an industry effort to produce XML schemas for describing directory data. It would seem natural to package directory requests in XML (perhaps using schema information defined by DSML) and to receive directory search results in XML (again possibly described by DSML schemas). The results could then be displayed using XSL style sheets to render views in a browser.

At the time of this writing, DSML is in its infancy. There are links to XML and DSML in Appendix H (book and CD). However, on the CD (under this chapter) you will find Visual XML tools from IBM (also see Appendix H for IBM's alphaWorks and Developer Works, either of which may have newer XML tools or DSML schema information for downloading). Using these tools, you could build your own XML encoding for directory data (we didn’t say it would be easy).

Improved Administration

In addition to sharing data across implementations (which also touches on administration), there will be improvements in general administration tools. For example, once you have linked three directory services together, wouldn’t it be nice to have a monitoring tool that showed utilization data and made corrective suggestions across all three?

Even before administration is improved across directory implementations, some improvements can be made to single vendor implementations. For example, when was the last time your directory service suggested that you schedule a backup, notified you of a curious pattern of authorization failures, or downloaded a fix and asked your permission to apply it?

Monitoring Tools

Monitoring can be done for anything in the following categories.

- **Performance**—identify hot spots, network connections that are over- or underutilized, classes of service for classes of users, or future needs based on statistical traffic growth analysis

- **Security**—identify unusual activity, patterns of authorization failures, individuals or programs that are sniffing around (See Figure 14.3.)

- **Unusual events**—heartbeat monitoring to ensure that services are up and stable, crash events, configuration errors, or long-term statistical changes in usage patterns (these may or may not be performance problems)

Monitoring tools can be integrated with management tools across vendor implementations to maximize customer value.
Identity Store

Identity on the Internet is a controversial issue: Some people want it, some people want to be anonymous, and most businesses want to identify their customers with certainty. Security issues abound on the Internet and corporate intranets. As Chapters 25 and 27 discuss, directory services can be used to store and manage identity in heterogeneous environments. (See Figure 14.4.) These mechanisms will likely make their way into more and more operating systems. In Novell’s Netware and Microsoft’s Windows 2000, identity is already very tightly tied to integrated directory services for their respective platforms. In the future, identity may span operating systems.

Device Management

In the good old days, devices were mechanical things that didn’t talk out of turn. They took instructions and returned results. Today, a router, for example, is a highly sophisticated computer. It can actively seek out policy to implement, such as quality of service (QoS). QoS is a way to prioritize traffic in networks, as for example, when the traffic generated by one class of users needs to be serviced ahead of other, lower priority or bulk traffic. An enterprise may have hundreds or thousands of these intelligent devices. Where are they going to find their policies? Do you have to pay some-

---

2. Links for Desktop Management Task Force (DMTF) descriptions for devices and management can be found in Appendix H.
one to go visit each one, load instructions, and come back later to make sure they are being followed? Policies and other device management information need to be centralized to lower the cost of administration. Centralized and always available yet secure access to data is the forte of a directory.

**ACID RASSS Everywhere**

While it would be great, we just can’t afford easy-to-manage ACID RASSS everywhere. There are definite costs and trade-offs associated with ACID, RASSS, certain performance characteristics, and manageability. Things do change, though, and as the costs of various pieces of technology drop, more and more directory services will look like super-databases or super-file systems or super-transaction monitors. Of course, this is a directory person’s point of view. A database person might say that directories are just growing up.

**Summary**

In this chapter we described where directory services might go in the future. It is a sure bet that directories will continue to add new function, while maintaining their current usefulness as common stores of data across systems. Some of this useful new function will be familiar from database, transaction processing, and file systems, such as levels of transactional support, integrated management, and storing new types of data, such as large objects.
The chapters in this section take the reader on a journey from the primitive services that directories provide up through the structure of directory servers. We examine how information is transferred to and from directory servers, how directory servers coordinate to present the illusion of a single service, and how directory information can be structured. This section presents how a directory service is made up of directory servers and how the servers interoperate to support a well-structured information tree.
Chapter 15
FUNDAMENTAL ENTERPRISE DIRECTORY SERVICES MANAGEMENT

This chapter is about the management of directory services and data. As we have seen, directories have a lot in common with databases, file systems, and even transaction processing monitors. But how do you manage the fundamental value proposition of a directory service?

The Political Dimension

The “glass house” staff and the “departmental staff” are frequently at war over issues such as change control, stability, auditing, service guarantees, and supported software and configurations. When organizing a new corporate-wide directory service, you will discover that you must navigate these political waters carefully, since your new directory service will cross the boundaries of many central and departmental organizations.

Every phase of directory planning, deployment, and management has a political aspect, so watch your attitude. Politics are not dirty, lowbrow, or the realm of dimwits and sycophants (well, not always), they are just a fact of life, and all aspects of directory management will be affected by the basic psychology of groups of people. People find comfortable niches, they understand the power structure, and they like things the way they are. A directory promises integration, easier and more efficient access to data, new ways to use data, and new ways to manage data across groups. (See Figure 15.1.) All of these things can upset comfortable people and change power boundaries.

1. The identity and definition problem that affects the term directory affects some of these other systems as well (see Chapter 14). For instance, the definition problem for the term transaction monitor is best described as follows: “In a contest for the least-well defined software term, TP monitor would be a tough contender.” 1993. Jim Gray and Andreas Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann.
By their nature, directories affect many suborganizations, divisions, groups, or other management units. The value proposition for directories centers on management and usage of data across a wide spectrum of sources and sinks and this can make directories a political football. The best way to prepare for this is to have a comprehensive plan that starts with how the new service is going to benefit each of the affected groups. Include all groups in the planning stage and let them have a say.

**Managing Directories**

You need to consider managing the fundamentals that make directory services useful as enterprise tools. The services that make a directory valuable also require some tending (and planning, debugging, and so forth). While some of the features and management of a directory are common between directories and other software services, some are unique. Following is a list of the fundamental directory services that need to be managed.

Enterprise data storage—how and what data to put in the directory, who is allowed to put data in, how new data will affect performance, how new data can be optimized for capacity, and data organization. Not all data are alike, so we need to categorize data and treat each specially.

Enterprise data retrieval—including how to efficiently mine data, what types of data searches are anticipated, who can access directory data, how much

---

2. See Chapter 29 for a case study showing the value of data integration through directory services, combined with the pain of getting groups of people to work toward a common goal.
resources they are allowed to use, and performance issues (search time and how it is affected by capacity, for example)

Enterprise data operations and management—backup and recovery, performance analysis, capacity planning, security auditing, and day-to-day operations and control. Enterprise data operations management is a bigger problem than a single directory server, or even a directory service. Data management includes coordinating information between directory services and other stores of data or at least directory services and other directory services. Data management also involves removing unused or orphaned data from the system.

The first thing you may notice about the list is that it doesn’t lend itself to an API reference or a sample program. This isn’t just a directory phenomenon. No amount of studying the LDAP C API or any other API will help you solve all enterprise-level problems. That’s one of the reasons we wanted to write a book that goes beyond APIs and discusses both technical and business issues related to directories.

**Enterprise Data Storage**

How data are stored is a major concern for large business environments. Just as the complexity of a program increases with its size, the complexity of data storage increases with the size of the organization or the amount of data stored. The increase in complexity can be nonlinear; that is, the storage problems of large organizations are immensely more complicated than those of smaller organizations. Instead of using a nightly backup to tape, large organizations classify hierarchies of data, and identify critical and noncritical systems. The storage problem (and other problems) for a small Web site that is infrequently visited pale in comparison to the problems encountered by an on-line business with significant on-line sales.

**Organizing Directory Data for Performance**

Depending on the directory implementation, you may be able to optimize data access for better response time or better system utilization. Some directory implementations are actually an LDAP veneer on top of a relational database (see Figure 15.2), so the rules for optimizing performance of a relational database can be applied, along with any rules that may make the veneer perform better.

Optimizing the performance of directories that are not implemented on top of a relational database requires consulting the vendor (or at least the technical references). Before embarking on a mission to optimize performance, you need to know that you have or will have a performance problem that needs to be solved.
An engineer once approached one of the authors with an optimization that removed thousands of instructions from the pathlength of an operation. Normally this would be cause for celebration, but the optimization was for a path that was exercised only during an administrative action and so was infrequently used. Sometimes, you have to step back and realize that not all performance gains are worth the time that it takes to discuss them.

Structuring the Directory Data

There are general rules about the structure of directory data that can improve performance. Performance can often be improved by understanding the critical paths that business applications take and optimizing the directory data model to match the data usage. That is, if you know that a critical application needs to search for data in a particular way, it may pay to take that information into account when designing your schema, directory configuration, or directory application.

However, if the critical application or applications make mostly ad hoc, or less patterned searches, optimizing the data storage model may not be as valuable. As an example, some directory vendors recommend that organizational unit objects not be nested too deeply, because the internal directory search algorithms react poorly to that structuring. In fact, to make searches more efficient, at least one directory vendor builds a parallel structure to the OID naming scheme (object identifiers are described in Chapter 16). It isn’t immediately obvious why this makes searches more efficient, until you know that OIDs are variable length, which can affect search algo-
rithms. Some surprising things can affect performance, and most of them are directory specific.

For more information on designing schemas for better performance see Chapter 16.

**Who Puts Data in the Directory Versus Who Gets Data Out?**

What would happen if you built an interface that allowed users to create new objects in the directory? Of course if depends on who your users are, what applications they use, and how those applications are controlled. If users can only create directory objects as a byproduct of using a critical business application, access is controlled by that application. For example, a Web application may keep user profiles in a directory, so the Web application can be responsible for creating the profiles, and for periodically getting rid of them as they age and are not used.

If users were able to create their own directory objects, who would get rid of them, or would the directory simply grow forever? Chances are, you want to control who has authority to add objects and to change objects. Instead of spreading this authority around, you should keep it in as few places as is practical.

**Categorizing Directory Data**

It's pretty well agreed on that data can be categorized by importance. What is less obvious is that there are other categorizations of data in a directory. For example, replication occurs on a schedule. It is very possible that some data should be replicated as soon as it changes while other data can be changed in batches. An example is removing a user object. Suppose someone is fired. It is desirable to remove that person's access to all systems instantly, but the user object is in a directory and the replication is set to occur only every half-hour. It's an extreme example, but a fired person might conceivably have a half-hour to wreak havoc in your enterprise if you have implemented a single sign-on scheme that is directory based, but you failed to take proper precautions.

Chapter 19 covers replication.
Enterprise Data Retrieval

As a directory administrator, you care about several aspects of directory data retrieval.

- How you enable the programming staff to get at data
- Whether end-users view directory data or it is hidden within applications that exploit the directory
- How you can make data access cost-efficient
- How you can make directory data access perform well enough to meet the requirements (obviously this depends on what the requirements are, which in turn requires you to know something about the future)

Let’s look at these issues within the context of a few simple LDAP API invocations.

White Pages Retrieval— If It’s That Easy, There Must be a Catch

You already know about white pages. Sometimes, people talk of constructing white pages with a directory service. By this, they usually mean building an employee directory (see Chapter 20) or a customer directory. However, white pages can be thought of more generically. Any time you look for one specific object in the directory to get some of its attributes you are using white pages lookup.

In the LDAP C API, you usually perform a white pages lookup by specifying the exact object (or entry) in the directory that you want to access. You perform the search and examine the attributes for that particular object. For example, to find a specific object, you might use the following LDAP search API. The bind operation that set a root distinguished name, or DN, to begin the search is not shown.)

```
rc = ldap_search_s(ld, mydn, LDAP_SCOPE_SUBTREE, L"(&(objectclass=serviceConnectionPoint) (cn=AWLEnterpriseDirectories)"), attributetoreturn, FALSE, &result);
```

3. It isn’t terribly important to know all of the parameters for this example to work, but you can look up the ldap_search API in several of the books on your CD, or look in Appendix A.
This LDAP search looks for a specific serviceConnectionPoint object (a pre-defined object class in the Windows 2000 schema that is used to advertise services) with the common name (cn) AWLEnterpriseDirectories. If this search is successful we can then look at the attributes and values for this object.

The preceding example is so simple (once you know that particular API) it is tempting to think that real life can be simple too. But what happens if every customer conducts that same search every time they log in to your Web site? Remember that we are talking about an enterprise: The Web site isn’t one HTTP server sitting in someone’s office, it’s a conglomeration of dozens of web servers. (See Figure 15.3.) If you have dozens of Web servers with hundreds or even thousands of concurrent customers, and logic being started by the Web server (CGI, application server, or whatever) looks for the AWLEnterpriseDirectories object, is it still a simple problem? No, it isn’t.4

---

4. For a discussion on the complexities of building and managing large-scale Web farms, see the IBM Redbook, Olympic-Caliber Computing, IBM number SG24-4279-00, available via www.redbooks.ibm.com. While dated, it still has some interesting information.
The C LDAP search API made it easy to look for data in the way that a steering wheel makes it easy to drive a car. However, it isn’t easy to move 10,000 people with your car, no matter how easy it is to drive, and it isn’t easy to find directory data if you also have to worry about scale, performance, and management. Enterprises can often have bigger and more complicated concerns than small organizations, so you, as an enterprise software engineer, IT leader, or manager, need more than an API reference—you need a full mental toolkit of enterprise software techniques. One of those techniques is to see performance bottlenecks before they happen, because they will happen on the busiest day of the year when your business must not fail.

So how do we solve the simple-yet-complex problem of the LDAP search being executed far too many times for any one server to handle? The question almost answers itself: have more than one directory server to serve the Web servers (or application servers that the Web servers are using). For directories, replication may be the answer, so that you can have a set of directory servers, each of which has a copy of the data that is required (see Chapter 19).

There are many other types of performance optimizations that you can consider. Some LDAP extensions, for example, are implemented as controls to aid in searching and sorting data on the server or on the client, whichever makes more sense for the application (and system and network impact).

Yellow Pages Retrieval—An Even Bigger Catch

Recall that a yellow pages search involves looking up a topic and getting back objects that relate to that topic. Let’s take the LDAP search that we used before and modify it slightly:

```c
rc = ldap_search_s(ld,
                   mydn,
                   LDAP_SCOPE_SUBTREE,
                   L"(objectclass=serviceConnectionPoint)",
                   attributetoreturn,
                   FALSE,
                   &result);
```

Now, instead of looking for a specific named object, we are looking for any object of objectclass serviceConnectionPoint. What if this logic were executed every time a customer came to your Web site? What is the relative cost of the search for a specific named object compared to searching for any object of a particular type? Imagine a search for a specific user object called “John Doe” and another search for all user objects.
Clearly, what you search for, how you search for it, how many of the objects exist, and how the directory service is constructed (its internal algorithms as well as its configuration) make a huge difference in the ability of your design to handle enterprise loads. Does it matter for a small organization with underutilized hardware where performance isn’t an issue? Of course not, but it matters a great deal for organizations trying to do real business on the Internet. It will be worth your while to investigate tools that forecast performance over a range of utilization scenarios. Sometimes tools such as these are available from your directory vendor. Other times you will have to write test tools yourself—loading a million objects into your directory and performing tests with many simulated concurrent users, for example.

Suppose that the directory objects you are retrieving are not open to the public but are protected by access control (as any vital data probably would be protected). Access control list checking for every object that you try to access can increase the system resources required to perform the search.

Many factors affect whether your usage of directory data retrieval is acceptable. All of the factors will come to your mind when you take the enterprise principles to heart and apply your toolkit of enterprise software techniques (for a refresher, look for RASSS in Chapter 2 and remember to think big). Also, find out how your directory works inside.

Here is another yellow pages lookup, in which we have a little more information in the search filter:

```c
rc = ldap_search_s(ld,
    mydn,
    LDAP_SCOPE_SUBTREE,
    L"(&objectclass=serviceConnectionPoint)(cn=AWL*)",
    attributetoreturn,
    FALSE,
    FALSE,
    &result);
```

In this yellow pages lookup, we modified our search to look for all service-ConnectionPoint objects with a common name (cn) that starts with AWL. We narrowed the search down a little by looking for objects that satisfy one hard criterion (object class) and one soft criterion (a partial name). This may perform better than simply looking for anything that starts with AWL, but how do you know that for sure? The answer depends on the internal algorithms of the directory server, whether the server has to contact other servers, and even the configuration settings (some directory servers allow you to create indexes the way you would on a database). Most of these performance parameters are not standardized, so only your directory vendor can give you answers.
All of the white pages enterprise concerns apply to yellow pages lookups. In addition, yellow pages lookups are easier to abuse. Looking for all of any type of object across an enterprise name space is likely to consume resources on a number of servers. Imagine looking for all employees of IBM, worldwide, whose last name starts with H. It isn’t a big deal if it’s only done once in a while, or if there is only one properly indexed server where this information resides, but it is a very big deal if it is done often or if the configuration isn’t a single server.5

It also might be a big deal if just anyone is allowed to perform a wide-open yellow pages search, such as looking for anything of any type. An errant program could sit in a loop doing this search and chewing up the resources of servers worldwide. In fact, what’s to stop someone with impure motives from setting up such a program and preventing legitimate users from accessing your on-line store (called a denial of service attack)? A major concern should be making the right data available to the right clients. However, the converse is also true: Make sure the wrong people are kept out.

Here is an LDAP search that is a request to return every object in the directory, no matter what it is (you could call this the mother of all yellow pages searches). You would not build an interface that would allow a customer to do this from a Web browser. If you build such an interface, you should consider using audit trails and automated procedures to find people who abuse it, and perhaps classes of service so that properly authorized clients have more “power.”

```c
rc = ldap_search_s(ld,
    mydn,
    LDAP_SCOPE_SUBTREE,
    L"(objectclass=*)",
    attributetoreturn,
    FALSE,
    &result);
```

Is LDAP Deficient?

We’ve been using LDAP C API examples and showing some potentially bad practices, or at least pitfalls to avoid. The point isn’t that the LDAP C API has more “holes” in it than any other API—the point is that you need to understand how APIs are actually implemented in directory servers. That is as true for the LDAP C API as it is for Java and JNDI, or Microsoft’s ADSI, or any other form of directory access.

5. This is another example of that ancient proverb of computer science: If it isn’t in a loop, don’t care; if it is in a loop, it will bite you. Updated this becomes: If it is in a loop driven by Internet clients, it will bite you hard and someone will try to abuse it.
Enterprise Data Operations and Management

Managing data is a big problem in typical enterprises. They often have more data than they can handle or that they can manipulate effectively. This is one of the problems that directories can help solve, but directories also introduce another kind of data that has to be managed.

How can a directory help solve data management problems? Think of a typical relational database. How would you organize computer users in such a database? One table? One table per country, since countries often have different laws concerning aspects of employment? Or a multitable design with references to a type of table per country? All of these ideas are constrained by the relational aspects of table design; it's just not easy to manipulate the data into a hierarchical view (that's why it's not called a hierarchical database).

Directories are fundamentally hierarchical in nature. Instead of defining a table with well-known columns, we define a tree of containers (the directory information tree—DIT—and the schema). Containers may hold objects of many types. For example, you might define a nested structure of organizational unit containers to mirror your company's organizational structure (for more OU options, see Chapters 17 and 25).

A hierarchical model can allow you to build a structure for data that models the real structure of an organization, or of whatever the data represent. For administration purposes, a hierarchical model can allow you to delegate control of the data on a sub-tree basis. Using the OU example, if you create a big OU that represents an entire group and nested OUs inside it to represent subgroups, you can bestow administrative authority that matches administrative responsibility. That is, someone at the top OU can do anything, but someone in a nested OU only has the authority to create and delete users (as an example) within that nested grouping.

The Enterprise Locator Service

Directories are wonderful locator services. You can find anything that advertises itself in a directory: users, applications, servers, even other directories. Used this way, directories can be the glue that binds together many disparate things. How do clients find a database, or a queue? If the database manager or queue manager advertised in the directory, he or she could be located using standard LDAP interfaces.

The directory locator service can have a profound affect on the operations of many services in the enterprise and on the operations of the directory itself. The more resources that are located through the directory, the more important it becomes to identify directory utilization (throughput) and health (for example, heartbeats).
quickly and accurately. In the ideal world all things can be located in the directory, which makes the directory vital to the running of all clients and increases the necessity that the directory service be highly manageable.

Management of directory services is, as you might suspect, not standardized. Some directory servers have wonderful GUIs, performance tools, capacity tools, troubleshooting tools, and development tools. Others do not, or have varying degrees of each.

**Directory Management as Part of the Bigger Picture**

Directory management is similar to the management of a database system. Some of the tools are similar (backup/restore, capacity planning, etc.). In particular, the running of a directory service has many of the same operational characteristics as running any service. These include knowing when the service is healthy and how to spot trouble.

From an operational perspective, any service must be manageable. In complex environments, manageability is a prerequisite to considering a service for production. Suppose your directory service has its own unique GUI and supports no special hooks or standards (such as SNMP) to integrate it into commonly available management tools (such as those from Tivoli or Computer Associates). Then you will have to build special operational procedures around that directory and you may incur hidden costs. Figure 15.4 is an example of a Tivoli console for managing several types of services, including LDAP and DNS directories.

**Summary**

At a fundamental level, directories just store, retrieve, and manage data. This applies to file systems and transaction processing systems as well; what defines directories are their lower-level aspects in managing enterprise data.

Enterprise software solutions are not usually easy. If they were, an API reference would be all you need. Instead, enterprises usually consist of multiple data solutions built up over some years; each has its own vital data that needs to be used effectively. Directories can help by providing not just location services, white pages, and yellow pages, but also by being an enterprise’s information infrastructure. Directories can help tie together disparate data sources. But you have to be careful—remember the enterprise principles and look beyond a single service toward managing data across the enterprise.

6. For more information on IBM/Tivoli management products, see http://www.tivoli.com.
Figure O.4  Managing multiple services
A directory service always has rules that govern the format of information content. These rules are called the *schema*. Defining schema for applications that use the directory service is one of the most important parts of using the directory service. This chapter discusses schema for the directory service. Chapter 17 discusses more details on building a schema for an application.

The Aspects of Directory Service

There are five basic concepts that pertain to a directory service. A directory service has

1. an administration model
2. a database implementation
3. an access protocol
4. a programming interface
5. a schema

While access protocol, API, and database implementation are usually fixed by the directory service implementation and, in fact, are features of the implementation, schema and specifics of the administration of content in the directory service are typically user or application controllable. Indeed, one of the features that makes a directory service useful to so many applications is that the application is free to define the format of the data the application stores in the directory service.

Being able to control the format of data in the directory allows applications to define an information layout that is natural for the application. This facilitates two aspects of using a directory service: ease of use and performance. Performance can be enhanced by co-locating information that is typically asked for together while ease of use is enhanced by a logical data organization that helps the user navigate through the information stored.
Even if agreements are made across the other aspects of a directory service—protocol, programming interface, database, and administration—agreement on schema among cooperating applications is still required so that the information stored in the directory can be re-used. In fact, the most important aspects of a directory service to agree on are access protocol and schema. These define the level of interoperability that can be obtained with the information stored in the directory service, because these aspects define the format of the information and the way to access that information.

The Function of Schema

When thinking about schema, think about modeling information. Armed with information about the data, a directory can parse, analyze, interpret, and act on data. The schema for data in the directory service is the information model for data that is accessible from the directory service.

What Things Are

A directory service makes a wealth of information available for applications to exploit just like a highway system makes a country available for people to enjoy. Without a road map, the roads appear to be a confused, tangled mess of concrete and asphalt, hard to navigate and impossible to use.

Schema is to the directory service what a road map is to the highway system. A schema defines the format of information that is found in the directory service. With knowledge about the information content, an application can interpret, act on, and even modify the information in the directory while allowing other applications to make use of the information.

A directory service provides access to a large amount of information. The schema for information in the directory service provides a description of what this information looks like. A road map legend describes the visual keys that show what things are. A directory service schema goes even further than a legend on a road map: it defines the format of all the information contained in the directory, allowing applications to interpret and act on this information.

The definition of the schema for information in the directory is similar to the definition of the data model for traditional database applications. But unlike many traditional database applications, information contained in a directory is used by multiple applications, distributed across many systems in the enterprise. These applications are typically related in their need for common information stored in the directory.
Agreement on schema constitutes an agreement on a common data model for the common information in the directory that multiple applications use.

In a generic directory service in formation is stored in a hierarchy of entries. Each entry has a name, called its distinguished name or “dn.” Each entry contains an objectclass attribute that describes the type of the entry. Each entry contains a set of name and value pairs called attributes. Attributes can have one or more values. (See Figure 16.1.)

**Where Things Reside**

Just as a road map shows cities, towns, and recreational areas, a schema for the directory service defines where information can be located in the directory service. Information in the directory service is structured in a hierarchical format, however: Information in each entry can be used to reference other locations in the directory. This allows related information to be referenced either by its name in the directory or by a pointer (which is the referred to entry’s full name) from some other entry in the directory service. This pointer, really just the name of the related entry in the name space, is called a dn-pointer. (See Figure 16.2.)

---

**Figure P.1** Format of an entry in the directory
What Is Mandatory and What Is Optional

Information stored in the directory service can take many forms. It is common that information about something is at times more generic or less specific than at other times. The information model for the directory service allows for this by defining information as either mandatory or optional. Mandatory information always appears in entries conforming to the schema while optional information may not necessarily appear. (See Figure 16.3.)

Allowing information to take on a variable form means the directory service can model generic entities as well as specific forms of those entities. Applications can treat the entities in as specific a form as is desired. If only the mandatory information is required by the application, less information can be administered. More specific information can be specified if the application or user desires. The additional information would be more data to administer but may help clarify the information stored in the directory service.
The Role of Schema

The schema that is used for storing information into the directory service affects the performance, usability, and extensibility of the application. The schema used by the application also affects much of the application’s implementation because it defines the set of information used by the application as well the order in which the information is accessed and updated.

Performance

Schema defines the grouping of information stored in the directory service and used by the application. This information grouping affects what set of directory entries must be updated for a given change or addition to the directory service. If the definition of the location of an application requires updates to four entries in the directory, then defining this location will require four separate updates to the directory service. This is because the set of add and change operations that can be performed against a directory service are defined to operate on a single entry at a time.

Additions and changes to the directory service are individually atomic. This means that each update to a specific entry in the directory is guaranteed either to complete or to fail, but not to partially complete. However, if two or more updates are required to the directory service, these updates must be synchronized by the application. Synchronization of
these updates requires additional design and implementation in the application. Multiple updates are also exactly that—multiple steps needed in order to complete a single operation, from the point of view of the application.

Wherever possible in the design of the information model that the application will use, information that will be used together or updated regularly should be colocated or located in a subtree of entries in the directory. This will reduce the number of directory query and update operations required when information in the directory is accessed or changed. Reducing the number of operations simplifies the application’s design and helps to improve the application’s performance. (Refer to Chapter 11 for more information on the performance of directory services.)

Usability

If the information used by an application is scattered across many entries in the directory, users will have a difficult time understanding the intent of the application and how it operates. Understanding what an application will and will not do is often critical to the usability of the application. Since the schema defines what information is stored where in the directory service, this information model is central to the usability of the information both with the application’s help and outside of the application’s help.

There are times when direct access to the information in the directory service is desirable. This can happen if multiple applications share a common set of schema, or when a generic directory service “browser” is used to view and update information in the directory service. If the information model is well designed, as information deeper and deeper in the directory tree is viewed, more and more specific information becomes available. This so-called drill-down support is only possible if the schema used to store information in the directory service is set up with this in mind.

It is possible to create more complex structures of directory entries than the basic hierarchy defined by the directory service itself. This is done by storing a distinguished name of a related directory entry in another directory entry. This stored distinguished name is sometimes called a dn-pointer because it is used to point to a location in the directory service that contains related information. The dn-pointers allow very complex information models to be built and maintained by the directory service. While dn-pointers are very powerful tools for creating the information model used by an application, their use can make the information model harder to understand. This affects the usability of the information stored in the directory service.

A logical organization of information makes the schema usable by an application and understandable to that application’s users. The hierarchical organization of the directory service should guide the discovery of more and more specific information (drill-
down). Careful use of dn-pointers enables directory entries to refer to one another, tying multiple entries together based on their relationship to each other.

**Extensibility**

Applications are rarely defined and implemented once, then left unchanged. Rather, successful applications initially serve some useful purpose and are then continually extended to support more uses. Successful applications must be extensible without major disruption to already working environments. In order to support the extensibility of an application, the information stored in the directory service by that application must itself be extensible.

A complete unload and reload of information stored in the directory service by an application can result in a large disruption in service for the application. This is undesirable from both the user and the application’s point of view, and should be avoided whenever possible. For this reason, the schema defined for an application should account for future extensions in an application’s design.

Version indicators, optional attributes, and object “inheritance” can be used in defining schema to allow for future extensions. The schema used by an application, because it defines the data model for information in the directory service, will affect the extensibility of the information as well as the application. Thus, extensibility should be designed into the schema defined for the application. (Refer to Chapter 13 for more information on managing the life cycle of applications.)

**Administration**

As discussed earlier, it is sometimes desirable for directory and application administrators to access the directory service using a generic tool called a directory service browser. A directory service browser presents the tree of information stored in the directory service. It also presents the information contained within one or more entries in the directory service in a form that is directly related to the schema that defines this information’s format.

Sophisticated directory service browsers use the schema to display the information contained in the directory in different ways, depending on its format. The schema for the information helps the “browser” present the information in the directory service in a way that is more understandable and easier to update than if the information were merely presented as a bunch of name/value pairs. When information in the directory information tree is grouped so that deeper entries in the tree describe more and more detailed information, the directory information tree will guide administrators in finding the set of information they want.
When multiple applications agree on a common schema, administration of both applications is simplified. Administration can be performed just once to affect both applications. This situation occurs most often with the definition of user and group information for each of the applications. If a common user and group model (user and group schema) are agreed on, then user and group administration can be performed from either application’s administration model or from a directory service browser.

Since the directory service is a shared resource, it should be treated and exploited as such. The schema defined for applications will necessarily contain application-specific information. However, wherever possible, common definitions should be agreed on and used across applications so that performance, usability, extensibility, and administration benefit from the schema definition.

**Schema in LDAP**

Schema in LDAP has been associated with the version of the LDAP protocol that is implemented or being used. Unfortunately, these two aspects of the directory service have little to do with each other. Schema in LDAP has typically been referred to as either *LDAP V2 schema* or *LDAP V3 schema*. The truth is, the schema definition should be (and, in fact, is) independent of access protocol. The definition of schema in LDAP should be tied closer to the directory service server and database implementations in terms of what features are available for schema designers to use.

**X.500**

Just as the LDAP protocol is defined as a simpler way to access a directory service than the X.500-defined directory access protocol (DAP), the schema definitions for directory servers that only support the LDAP protocol are based on the X.500 data model and schema.¹ The X.500 data model consists of a hierarchy of entries called the *directory information tree* (DIT). Each entry in the DIT contains one or more attributes. Each attribute within an entry has a type (called the *attribute type*), and one or more values. The set of attributes that are allowed to be held in any particular entry are defined by the entry’s object class. The object class of an entry defines what attribute types must and what attribute types may appear in an entry.

As more and more applications begin to exploit directory service, more and more sophisticated aspects of the X.500 data model will be required. For most applications, the basic concepts of syntaxes, matching rules, attribute types, object classes, object

---

¹ X.500 family of standards (ITU X Series recommendations)
class hierarchy, and directory information hierarchy are sufficient to cover the application’s data modeling requirements.

This bears repeating: The aspects of schema in LDAP are not new. Schema in LDAP uses the most common aspects of the more general X.500 data model. Using the characteristics of syntax, matching rules, attribute types, object classes, object class hierarchy, and directory information hierarchy, complex generic and application-specific information can be modeled and stored in the directory service.

Syntaxes

Syntax defines the format for individual values stored in the directory service. The syntax for a value defines the type of the information. As an analogy, programming languages define the basic set of types that can be used in the language. In the C programming language, for example, basic types such as int, float, double, long, and char are defined. In a directory service, the basic set of types that are available are typically defined by the server implementation that is chosen. Even the most sophisticated directory servers usually define the basic set of syntaxes that are supported by the server implementation. For the most commonly used syntaxes, the LDAP protocol defines a string form for information stored using the syntax. This facilitates easy retrieval and parsing of the information by applications using the LDAP protocol.

Code Example 16.1 shows the format in which directory servers must publish the syntaxes they support when the server supports the LDAP Version 3 protocol, as described in IETF RFC 2252 (see Appendix H). In the code example, a syntax is described by an object identifier and a description. The object identifier is then used in the definition of attribute types (see the section Attribute Types in this chapter).

The most common syntax types supported by servers that support the LDAP protocol (and thus make their information available as part of the directory service), are string syntax types (including DirectoryString and IA5String), integer, boolean, generalizedTime, and binary. There are a multitude of other syntaxes defined, but

```
SyntaxDescription = (  
  <numeric object identifier>  
  [ DESC <quoted character string> ]  
)

Example:  
  ( 1.3.6.1.4.1.1466.115.121.1.15 DESC 'Directory String' )
```

Code Example P.1  Definition of syntax specification from IETF RFC 2252
these tend to become more and more specific to the application for which they are defined rather than of general use in modeling information in the directory service.

The string syntaxes allow information that is defined as a set of written information (like a name, address, title, host name, user ID, description, caption, etc.) to be modeled as information in the directory service. String syntaxes are closely tied with their matching rules and generally fall into two categories based on the style of string matching required: case-exact matching, where the upper/lowercase character values are considered distinct, and case-ignore matching, where the upper/lowercase character values are considered the same.

The directory service is a shared resource, accessible from many operating systems and environments. Because of this, special care must be taken to store strings in the directory service in a format that will be readable by everyone—or at least defined in enough detail so that the strings can be converted to a format usable by the environment. String formats on operating platforms include single-byte ASCII, single-byte EBCDIC, double-byte EBCDIC, and even UTF-8 and Unicode. Early directory service implementations expected strings in ASCII format and returned strings in ASCII format. This format is more strictly defined as the IA5 character set, which is usually termed 7-bit ASCII. The limitations of this character set are well known, so the new directory service implementations support strings in UTF-8 format. UTF-8 is a multibyte character set with the useful characteristics that no embedded NULL (0x00) bytes will appear in a UTF-8 string and the single-byte UTF-8 characters are exactly the 7-bit ASCII characters. There is a straightforward transformation between Unicode (UCS-2) characters and UTF-8 characters. String data in the LDAP version 3 protocol is now assumed to be in UTF-8 format, and while directory client programming interfaces may assist the local operating platforms in performing the necessary conversions, communication with the directory service for string data is in UTF-8 format.

The integer and boolean syntaxes are related in that they both allow a numeric form of information to be modeled in the directory service. Integer values define numbers stored in the directory service and can be positive or negative, and their range is usually determined by the server implementation. Boolean values take on values 1 (true) or 0 (false). In some server implementations, integer and boolean syntax information is stored as string data since each has a string form that is used over the LDAP protocol. The string form for an integer is the string of decimal digits that represent the number (with a possible hyphen as the first character). The string form for a boolean value is either the string “TRUE” or the string “FALSE,” depending on the value.

The generalizedTime syntax is used to represent timestamps in the directory service. Timestamps are useful in many situations for logging the add, last-modified, or last-accessed time related to an entry in the directory service. Security applications that use the directory service use timestamps for noting the expiration time for security information. The generalizedTime syntax is preferred to the older UTCTime syntax.
because the generalizedTime syntax uses a four-digit year specification, whereas the UTCTime syntax uses only a two-digit year specification. The string form generalizedTime values is defined by the X.208 standard.\(^2\)

When no other syntax matches the type of information to be stored in the directory service, the binary syntax is available. Binary syntax provides the least specification of the information content. Binary values typically fall into the category of being very large or of a form that is meant to be opaque to all applications other than the application that is used to access and manage the values. Consider the other syntaxes supported by server implementations first before choosing the binary syntax for modeling an application’s information stored in the directory service. Note that storage of certificates and (small) photos in the directory service is a well-known and accepted use of a binary syntax.

Additional syntaxes are defined in some server implementations. These syntaxes typically represent structured types (information where each value in the directory service contains multiple pieces of information that is all related). While this form of data modeling sometimes cannot be avoided, its use causes the information to become more application-specific and typically harder to use.

**Matching Rules**

Matching rules are closely related to the syntax of values stored in the directory service. The most complex matching rules are related to the string syntaxes and the less-commonly used syntaxes that represent structured types in the directory service. Examples of matching rules are caseExactMatch, caseExactOrderingMatch, caseIgnoreMatch, and integerMatch.

Code Example 16.2 shows the format in which directory servers must publish the matching rules they support when the server supports the LDAP Version 3 protocol, as described in IETF RFC 2252 (see Appendix H). In the code example, a matching rule is described by an object identifier, a description, an indicator of whether or not the matching rule is now obsolete, and the syntax with which the matching rule corresponds. The object identifier for the matching rule is then used in the definition of attribute types (see the section Attribute Types later in this chapter).

Matching rules are categorized by whether the rule supports equality matching, substrings matching, or ordering of values. Equality matching provides only a true/false indication of whether two values match or not. Substring matching allows for substrings of attribute values to be matched. Ordered matching provides an indica-

---

\(^2\) X.208 standard (ITU X Series recommendations)
tion of less than, equal to, or greater than, and is useful when queries are performed for information below or above a certain value.

Like syntax types, the set of matching rules supported is typically fixed by the directory server implementation that is used to provide the directory service. As directory service server implementations mature, syntaxes as well as matching rules may become extensible through programmable exits offered by the server implementation.

**Attribute Types**

The basic unit of information that is accessible over the LDAP protocol is the value(s) associated with attribute types. Attribute types, or attributes, as they are commonly referred to, define values that can be stored in the directory service. The schema definition for an attribute defines the name of the attribute type, whether the attribute type is single- or multivalued, the syntax associated with the attribute type, and any alias names that might be used to refer to the same attribute type.

Code Example 16.3 shows the format in which directory servers must publish the attribute types they support when the server supports the LDAP Version 3 protocol, as described in IETF RFC 2252 (see Appendix H). In the code example, an attribute type is described by an object identifier, a description, an indicator of whether or not the attribute type rule is now obsolete, the syntax, and the matching rules associated with the attribute type, as well as other indicators of the usage of the attribute. Unlike the X.500 DAP protocol where object identifiers are used exclusively, the name of the attribute type is used over the LDAP protocol, making interpretation of the LDAP protocol simpler in practice.
Each attribute type in the directory service has a name. This textual name is used as the preferred way to refer to the attribute type when communicating over the LDAP protocol. An attribute type should have an object identifier (OID) (dotted decimal numeric value) associated with it. This OID will guarantee uniqueness of the attribute type definition. The LDAP protocol supports referring to attribute types by OID as well as name, but this is not the preferred usage because OIDs are cumbersome and less user-friendly.

Part of an attribute type’s definition is the syntax for the attribute values associated with the attribute type. All values for the attribute type must conform to the rules that define the syntax for the attribute type.

An attribute type is also defined as single- or multivalued. Single-valued attribute types are defined such that only a single value can be assigned to the attribute type within any particular entry in the directory service. Multivalued attributes allow multiple values to be assigned to an attribute type within an entry in the directory service. It is important to note that the order of the multiple values is not significant. The values are treated as a set, not a list. This implies that individual attribute values

<table>
<thead>
<tr>
<th>AttributeTypeDescription</th>
</tr>
</thead>
<tbody>
<tr>
<td>`( &lt;numeric object identifier&gt;</td>
</tr>
<tr>
<td>[ NAME ( &lt;quoted character string&gt; ) ]</td>
</tr>
<tr>
<td>[ DESC &lt;quoted character string&gt; ]</td>
</tr>
<tr>
<td>[ OBSOLETE ]</td>
</tr>
<tr>
<td>[ SUP &lt;AttributeType name or OID&gt; ]</td>
</tr>
<tr>
<td>[ EQUALITY &lt;MatchingRule name or OID&gt; ]</td>
</tr>
<tr>
<td>[ ORDERING &lt;MatchingRule name or OID&gt; ]</td>
</tr>
<tr>
<td>[ SUBSTR &lt;MatchingRule name or OID&gt; ]</td>
</tr>
<tr>
<td>[ SYNTAX &lt;numeric object identifier [{suggested length}]] ]</td>
</tr>
<tr>
<td>[ SINGLE-VALUE ]</td>
</tr>
<tr>
<td>[ COLLECTIVE ]</td>
</tr>
<tr>
<td>[ NO-USER-MODIFICATION ]</td>
</tr>
<tr>
<td>[ USAGE &lt; userApplications</td>
</tr>
</tbody>
</table>

Example:
```
( 1.3.18.0.2.1000.1.1.16.4.1 NAME ( 'aSerialNumber' 'aSerNo' )
  DESC 'Company serial number' EQUALITY 'caseIgnoreMatch'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 SINGLE-VALUE
  USAGE userApplications )
```
in an attribute type cannot be associated with values for another attribute type within the same entry (value 2 of attribute A cannot be associated with value 2 of attribute B). Multiple attribute values constitute an unordered set of values. In most cases, if the syntax type of the attribute type is boolean or integer the attribute type will be defined as single-valued.

**Object Classes**

*Object classes* group sets of attribute types together to form structures of information that can be stored in the directory service. There are three types of object classes: abstract, structural, and auxiliary, defined by the X.500 model. Of these, only structural and auxiliary object classes can be used to store information in the directory service.

Code Example 16.4 shows the format in which directory servers must publish the object classes they support when the server supports the LDAP Version 3 protocol, as described in IETF RFC 2252 (see Appendix H). In the code example, an object class is described by an object identifier, a description, an indicator of the superior object class if this object class is derived from another object class, whether or not the object class is now obsolete, the type of the object class (abstract, structural, or auxil-

```
ObjectClassDescription = ( 
    <numeric object identifier> 
    [ NAME (<quoted character strings> ) ] 
    [ DESC <quoted character string> ] 
    [ OBSOLETE ] 
    [ SUP <ObjectClass name or OID> ] 
    [ ABSTRACT | STRUCTURAL | AUXILIARY ] 
    [ MUST ( <AttributeType name or OID> 
           [ $ <AttributeType name or OID> ]* ) ] 
    [ MAY (<AttributeType name or OID> 
          [ $ <AttributeType name or OID> ]* ) ]
)

Example:
( 1.3.18.0.2.1000.1.1.16.6.1 NAME ( 'aEmployee' 'altEmployee' ) 
  DESC 'A new object class representing an employee' 
  SUP 'inetOrgPerson' 
  STRUCTURAL 
  MUST ( 'aSerialNumber' $ 'cn' $ 'sn' ) 
  MAY ( 'telephoneNumber' $ 'pagerNumber' )
)

**Code Example P.4**  Definition of object class specification from IETF RFC 2252
Each entry in the directory service has at least one object class associated with it. The object class defines the set of attribute types that must appear in the entry, as well as the set of attribute types that may appear (optionally) in the entry. Like attribute types, object classes have names, OIDs, and possibly alias names. Object classes define the structure of the information stored within entries in the directory service.

Abstract object classes are classes that are used to define other, more complex object class definitions. Entries cannot be added to the directory service that are defined to have the structure of an abstract object class.

Structural object classes define the format of the entries that will be stored in the directory service. In order for a new entry to be added to the directory service, it must be defined by a structural object class definition. As the name implies, structural object classes are used to define the structure of entries in the directory service.

Auxiliary object classes are the hardest object class type to understand. Auxiliary object classes define structures of information that can be attached to entries or added to existing entries in the directory service. An entry in the directory service must have at least one structural object class but can have zero or more auxiliary object classes associated it. Common uses for auxiliary object classes are for extending an existing entry's information with additional information, or, for associating a set of attribute types with multiple, different entries in the directory service.

**Schema Check**

Each time an entry in the directory service is added or modified, the directory service verifies that the proposed entry conforms to the schema that defines the syntax types, matching rules, attribute types, and object classes for the directory service. This process, called *schema check*, does the following.

- Verify that only a single structural object class is assigned to the entry. Multiple auxiliary object classes may also be assigned to the entry.
- Verify that the set of attribute types that is in the union of the required attribute types in each of the individual object classes are present within the entry (that is, contain at least one value).
- Verify that no attribute types exist in the entry that are not defined as part of the union of the required and optional attribute types defined for each of the individual object classes present within the entry.
For each attribute type defined to have one or more values in the entry, verify that each of the values for the attribute type conform to the syntax that defines the attribute type value format, and verify that single-valued attributes contain only a single value.

If all of these checks pass, then the entry is said to conform to the schema definition defined by the directory service.

**Object Class Hierarchy**

The schema for the directory service has some characteristics of an object-oriented system. All object classes are defined in an object class hierarchy. (See Figure 16.4.) This hierarchy has, as its root, an abstract object class called top. Additional object classes are defined by deriving a definition from some superior object class. The superior object class is sometimes called the parent object class. All object classes have the top object class as an ancestor in the object class derivation chain that defines the object class.

When a new object class is defined, it is derived from a superior or parent object class. The new object class contains all the attribute types defined in the parent object class in addition to all of the attribute types added as part of defining the new object class. This allows new object class definitions to build on previously defined object classes.

Using object class derivation helps define the schema for the directory service in two ways. First, each object class definition can be simpler to define because common attribute types across a number of entries can be defined as part of a parent object class and then included in all the derived object classes. Second, object class definitions can be defined as specialized forms of more generic objects. This allows applications to treat the object in either the more specialized form or the generic form.

The object class hierarchy is an important tool to use when defining the schema for the directory service. Careful attention to the use of abstract, structural, and auxiliary classes, and the object class hierarchy that is used can greatly simplify the schema definition as well as simplify the programs that must operate on the information in the directory service.
Directory Information Tree

Different from the object class hierarchy is the hierarchy of directory information. As described earlier, all entries in the directory service are defined as part of a hierarchy or tree structure. While these entries have object classes associated with them that describe the format of the entry, the entries themselves reside in a tree structure called the directory information tree (DIT). (See Figure 16.5.) (The DIT must not be confused with the object class hierarchy.)
With the definition of the LDAP Version 3 protocol, the directory service now supports the concept of schema publication. This allows an application to discover the schema associated with a portion of the DIT by following dn-pointers that are made available with every entry. For server implementations that support this, the schema can be discovered using the LDAP protocol itself. However, because the directory service will typically span multiple servers across the organization, different schema definitions may reside in different portions of the DIT.

LDAP Version 3 defines how schema will be published. The definition appears straightforward, but in practice it can be confusing to discover the schema that is being used in one portion of the DIT. The code in Code Example 16.5 shows how to find and retrieve the schema associated with some portion of the DIT. The subroutine defined here assumes that an input distinguished name refers to an existing entry.

Figure P.5  Example of a directory information tree

Schema in LDAP
in the directory service. Using this entry, the entry that contains the schema associated with this portion of the DIT is discovered by first following a dn-pointer (the subschemasubentry attribute type) in the entry and then retrieving the information in the entry pointed to by the dn-pointer.
The schema definitions accessible over the LDAP protocol are defined as a set of attribute values in two attribute types: attributeTypes and objectClasses. These attribute types themselves have their own syntax definitions, which conform to the descriptions of attribute type and object class shown earlier in this chapter. If a directory server does not support the publication of the schema that is active in the server, the subschemaSubentry attribute will not be present in the returned information.

Other Characteristics

A number of additional items are part of the schema definition for the directory service. The most important of these are DIT structure rules and naming constraints.

DIT Structure Rules

DIT structure rules are rules that define the set of object classes that can be used to define entries below other entries in the DIT. The most common example of this is the use of organization and organizationalUnit object classes. A common DIT structure rule is that organizationalUnit entries must appear below organization entries or organizationalUnit entries. This rule defines a consistent layout of information in the directory service and helps order what types of information will reside in what portions of the DIT.

Naming Constraints

Each entry in the directory service has a name, called its distinguished name. The distinguished name of an entry consists of the parent entry's name (parent in the DIT), along with a relative distinguished name (RDN), which identifies the entry below the parent entry. The relative distinguished name is really a subset of the attribute values that are defined as part of the entry information. One or more of the attribute values in an entry are designated as the naming attributes for the entry.

Naming constraints limit the set of attribute types that can be used in defining the relative distinguished name for the entry in the DIT. A common naming constraint is that organization entries must use the organization attribute type (o) as the naming attribute in the RDN. A second naming constraint that is often used is that organizationalUnit entries must use the organizationalUnit attribute type (ou) as the naming attribute in the RDN.
Like DIT structure rules, naming constraints are useful for bringing order to the DIT. Without these rules, the DIT can become unorganized and hard to interpret, weakening its usefulness, extensibility, search, and modification characteristics.

Name Space Layout

The layout of the name space is important because it will affect the distinguished names that are used when adding entries into the directory service as well as affect the ease with which the DIT can be traversed. As entries are defined deeper and deeper in the DIT, the information found in these entries should be more and more specialized. The upper levels of the name space should consist of entries that organize the set of information that is available in the directory service.

There are two general approaches to laying out information in the DIT. One model for name space layout is tied closely to the DNS names used in an enterprise. In this model, the upper levels of the DIT contain entries that are defined as dcObject or derived from dcObject. Each dcObject entry in the DIT is used as a container entry, similar to a directory/subdirectory structure in a file system. Other entries appear in the DIT underneath an appropriate dcObject entry at a level that is consistent with the associated DNS name.

Care must be exercised when using this model so that location transparency of applications can be retained. If a distinguished name made up of dcObject attribute types is part of the name of an entry and a computer system is also defined with the associated DNS name, then there is an implication that entries below the entry whose distinguished name is related to the DNS name represent information that is hosted on the computer system with the related DNS name. If this changes with an administrative action, the names of entries in the DIT may have to be changed as well.

The second model for name space layout is tied to organizational structures within organizations. Just about every organization is organized in a hierarchical fashion. Employees are part of organizational units; organizational units are parts of larger organizational units; and organizational units are part of organizations. In this model, entries defined as organization and organizationalUnit object classes are used to build the upper layers of the DIT. At the appropriate level in the DIT below the organization and organizationalUnit entries, entries representing people, applications, and other items are added.

Again, care must be taken when using the organization/organizationalUnit approach to naming information in the DIT. As people move from one organizational unit to another, or responsibilities change for applications which use the directory service for their information, changes may be required in the names of entries stored in the directory service.
ASN.1 (X.208) and X.501

As mentioned earlier, the origins of the data model that is defined for the directory service is X.500. In X.500, attribute types and object classes are defined in a declarative language called Abstract Syntax Notation 1 (ASN.1). ASN.1 is used to define attribute types, object classes, syntaxes, matching rules, and, in fact, complex structured information. The LDAP protocol itself is defined using ASN.1. The definitions of object classes and attribute types in LDAP can be viewed as simplifications of the definitions of these same concepts in X.500. As the definition of LDAP has progressed, the definition of schema has become more and more closely aligned with the definitions in X.500.

The ASN.1 definitions of object class and attribute type from X.500 are shown in Code Example 16.6.

```
name ATTRIBUTE ::= {
  WITH SYNTAX DirectoryString { ub-name }
  EQUALITY MATCHING RULE caseIgnoreMatch
  SUBSTRINGS MATCHING RULE caseIgnoreSubstringsMatch
  ID   id-at-name }

commonName ATTRIBUTE ::= {
  SUBTYPE OFname
  SYNTAX   DirectoryString {ub-common-name}
  ID       id-at-commonName }

top OBJECT-CLASS ::= {
  KIND        abstract
  MUST CONTAIN{ objectClass }
  ID          id-oc-top }

person OBJECT-CLASS ::= {
  SUBCLASS OF { top }
  MUST CONTAIN { commonName | surname }
  MAY CONTAIN { description |
                telephoneNumber |
                userPassword |
                seeAlso }
  ID          id-oc-person }
```

**Code Example P.6**  
ASN.1 definition of attribute type and object class
Note the similarity of the LDAP definitions of attribute type and object class with the definitions in X.500. In general, when you are trying to understand and define schema for the directory service, use the X.500 models as a guide.

**Summary**

This chapter presents the concept of schema for the directory service. The schema of the directory service is closely related to the data model of the directory service rooted in X.500 models. The schema defines the set of syntaxes, matching rules, attribute types, and object classes that can appear in the directory service. With these tools, complex schema definitions can be created that model many different types of information. By defining additional attribute types and object classes, and paying close attention to name space layout, you can make the directory service the central locating service for an organization. Definitions representing people, places, and things (computers, printers, servers, applications, file systems, distributed objects, and more) can all be modeled and stored in the directory service.
If an application is going to exploit the directory service, there is a strong possibility that the application will have some unique set of characteristics that must be modeled in the directory service. After all, applications are written to have unique features because this provides value to its users. This chapter discusses how to build a schema for an application.

**Custom Schemas**

**Why There Are So Many Schemas**

It seems that every application of the directory service finds it necessary to define a unique set of information that is stored in the directory service. Even if agreements are made on directory information model and access protocol, the format of the data stored in the directory service must be agreed upon before information can be shared and used by multiple applications. Definition of additional schemas seems counter to the notion of sharing information that is made available by the directory service.

One reason that multiple schemas have been, and will continue to be, defined is that applications always contain unique features. Many of these features rely on specific information that is managed by the application. The value-add capabilities that differentiate applications also result in extensions to any form of common schema that could be defined.

Another reason that multiple schemas have been defined is that the base, or common, schemas that existed previously are insufficient for the greatly expanded goals of employing a directory service across an organization. For example, the attributes defined for a person object class are very much related to a person's physical mailing address or electronic mail address. While this information is useful and necessary, it is not sufficient to cover the definition of a person within an organization, who typically has multiple identities on multiple computer systems, organization-specific information, and a number of roles and group memberships, all of which go into the definition of a person within an organization. The person object class defined in IETF RFC 2252...
(see Appendix H) is not sufficient to model these more specialized aspects of a person in an organization.

A third reason is that emerging applications are defining more and more complex and sophisticated uses of the directory service as a centralized management information server. The notions of policy management, network configuration and management, and systems configuration and management modeled in and done through the directory service imply additional data modeling for the information contained in the directory service.

Recent initiatives by the Distributed Management Task Force (DMTF) to exploit the capabilities of a directory service have produced information models like the Common Information Model (CIM) and Directory Enabled Networking (DEN). Each of these data models implies new schema definitions. The CIM standards have their roots in network management architectures and protocols, and are concerned with schema for managing networks and systems using a directory service. The DEN initiative is a cross-industry effort to standardize the set of directory information that is used for configuring and managing networks of routers and systems in an enterprise.

Custom Schema

The potential for information sharing between applications is a key feature of directory services. However, base schema definitions for user and group concepts in applications rarely suffice when used to model the users and groups that are defined and used by the applications. There is almost always a set of user-specific and group-specific information that is unique to the application itself.

In addition, each application tends to have its own unique set of information. This information may or may not be related to the user and group definitions. If not, the unique schema definitions must be defined in order to model the set of information that will be stored by and modifiable through the directory service.

When base schema definitions do not contain enough information or when existing schema definitions do not contain schema definitions that model the information for the application, then a custom schema should be designed, defined, and used.

Designing Custom Schema

Designing a schema for use in the directory service is similar to modeling the information that is managed by the application itself. In fact, entity-relationship modeling techniques can help determine just what schema definitions need to be defined so that the application can exploit the directory service.
Once a data model is defined for the information to be stored in the directory service, object classes and attribute types can be defined. These definitions should take into account the hierarchical structure of entries in the directory service. The definitions should use dn-pointers in handling the relationships defined in the data model. (The use of dn-pointers is described in Chapter 16.)

After the schema is defined, the functional characteristics of operating on instances of the schema should be considered. These operational characteristics include how entries will be added, deleted, modified, and moved around in the directory. Referential integrity issues regarding multiple-entry updates should be addressed by these functional characteristics. While some directory implementations support managing referential integrity between entries in the directory, others leave this to the directory exploiting application.

Defining the Data Model

Entity-relationship (E-R) modeling of the information to be stored in the directory service will help determine the set of entities to be operated on and the set of relationships to be defined that use dn-pointers. This E-R model will help define the set of attribute types and object classes that will be needed to model the information stored in the directory service. In addition, the E-R model will help define the object class hierarchy needed to define the schema.

A simple E-R model that models employees within an organization is shown in Figure 17.1. The entity called aPerson has a relationship with both the aDepartment and

![Figure Q.1 Entity-relationship model of aPerson, aGroup, and aDepartment]
the aGroup entities. The aPerson entity is specialized into two additional forms: aManager and aBoardMember. The aGroup entity is specialized into aDistGroup and aAccessGroup. The aGroup object class is marked as a generic entity, meaning that all instances of this entity, in practice, are either aDistGroup or aAccessGroup entities. The aPerson relationship with the aDepartment entity defines the aPerson as a member of the aDepartment entity. A relationship also exists between the aDepartment and the aManager entities that represents the manager associated with the department. (Refer to Chapter 24 for a case study on creating a corporate employee directory.)

Once an E-R model has been defined, how this model is represented in the directory service can be defined. The definition should exploit the hierarchical form for information in the directory service.

Exploiting the Hierarchical Name Space

Once a data model for the information to be stored in the directory service has been defined, how the information will be stored in the directory service must be defined. Just as in database design, where there is a difference between a logical data model and physical data model, there can be differences between the logical data model for the information in the directory service and the physical layout of the information stored in the directory service.

The performance and usability of the information stored in the directory service should be evaluated to determine how the information from the data model should be organized into the directory service name space. The most common access paths for the information should be streamlined to require a minimum number of interactions with the directory service in order to retrieve the information. This is usually accomplished by storing information in the same entry in the directory service and by using the hierarchical structure of the name space to allow multiple entries to be returned in a single search operation. Returning multiple entries is done by performing a search of the directory and requesting that all entries below the searched entry be returned in the search results. Thus, one use of the hierarchical name space of the directory service is to support grouping of related information for faster retrieval.

Another use of the hierarchical name space is to implement one of the relationships identified by the data model that defines the information to be stored in the directory service. A one-to-many relationship in the data model implies containment of one set of information under another entity, so the name space hierarchy could be used to implement this relationship.

A third use of the hierarchical name space is to support extracting more detailed information about an entity by traversing deeper and deeper into the directory service name space. In this use of the name space hierarchy, additional entries in the directory located below parent entries might contain more detailed information.
about the entity that is implemented by the parent entry in the namespace. This use of the hierarchy permits drill-down to retrieve more information about entities that are stored in the directory service.

Figure 17.2 shows how the directory service namespace hierarchy is used to implement the relationship between aPerson and aDepartment entities. In a containment relationship a person is member of (or contained in) a department within a company. The namespace is also used to implement a drill-down for obtaining more details about a person in the company: Extra information about the person is stored in an entry below the aPerson entry in the namespace hierarchy.

A fourth use of the hierarchical namespace is to implement the containment relationship between aDepartment entities. Just like the containment of aPerson entities is modeled using the namespace hierarchy, so is the containment of aDepartment entities within larger aDepartment entities.

In addition to modeling the containment relationships, the namespace hierarchy allows for efficient retrieval of entire departments. Furthermore, all information

---

**Figure 17.2** Containment relationship of aPerson entries under aDepartment entries
about a person can be gathered by performing a single subtree search on the aPerson

The hierarchical name space is a tool that should be exploited when implementing

Defining a Package

The next step in building a schema is to define the set of object classes and attributes

The definitions of object classes and attribute types do not contain version informa-

In order to address these issues, the set of object classes and attribute types that are
defined should be organized into a package. While a package need not have any rep-

In addition to grouping the set of attribute types and object classes, a description and

Further, a package should contain the list of other packages that the object classes
depend on. It is possible (and desirable) to re-use and extend attribute types and object classes when defining new schema definitions. If all

attribute types and object classes are organized into packages, they can be loaded
when a schema is loaded into a directory service. Without the concept of a package
and the list of dependent packages between them, deciding which attribute types and

object classes have been added to the schema in the directory service becomes a much more difficult task.

To ease administration, future extension, and management of the schema definitions that are enabled in the directory service, a package should be defined. The package should contain the set of attribute types and object classes that are used to implement the physical data model associated with the logical data model for the application.

The package in our example is called aCorporateSchema. Figure 17.3 shows this package definition. Note that the set of schema packages defined in the directory service could be stored in the directory service itself. A directory schema (object class and attribute types) to describe a package is shown in Code Example 17.1.

The location of package definitions in the hierarchical name space is arbitrary but should be organized with a container that groups all package entries together for easy lookup and retrieval.

---

**Figure Q.3** Package used to organize the set of attribute types and object classes in a schema
Defining Object Classes

The set of entities in our example that require object class definitions are aPerson, aManager, aBoardMember, aGroup, aDistGroup, aAccessGroup, and aDepartment. The aManager and aBoardMember object classes are specializations of aPerson, while aDistGroup and aAccessGroup are specialized forms of aGroup.

As described in Chapter 16, an object class can be defined to be abstract, structural, or auxiliary. The aGroup object class, in the example, is an abstract object class. No entries should appear in the directory name space with aGroup form: only aDistGroup and aAccessGroup object classes can be used. The aPerson, aManager, aBoardMember, and aDepartment object classes are all structural object classes, since entries of these forms are to be defined in the directory name space. The example does not require any auxiliary object classes. (See Code Example 17.2.)

To illustrate the use of auxiliary object classes, let's assume that the existing data model has been used for some time when the company decides to require all persons, groups, and departments to provide contact information in the form of a phone number or e-mail address. To extend the existing schema to cover this, an auxiliary object class can be defined, called aContactInfo. The object class would contain attributes for a phone number or e-mail address to serve as contact information.

Each object class in the schema has a set of attribute types that are required and a set of attribute types that are optional. The set of required attribute types are specified as mandatory attribute types while the set of attribute types that are not are noted as
optional attribute types. A special set of attribute types, naming attribute types, are not differentiated from other attribute types, but it is a good idea to keep track of which should be used when naming entries in the directory name space. These names will help users identify types of entries as they traverse the directory service looking for information.

Object classes that are already defined should be re-used whenever possible. When specialized forms of the same general type of information must be modeled, an existing object class can be used as the basis for defining the new object class. This allows the new object class to contain all the attribute types of the parent object class. The new object class can specify some of the existing attribute types that were optional in the base class as mandatory attribute types. The new object class can define additional attribute types as either mandatory or optional. In the example, aPerson extends organizationalPerson, while aGroup extends Group. The base class for the new object class definition is defined as the SUP object class in the specification of an object class.

Some directory service implementations allow greater control over the format of the directory name space by requiring object classes to specify the format of entries that can appear “above” the object class in the directory name space. This set of object classes, called the DIT superiors, should not be confused with the superior object class in the definition of the object class. The superior object class is the parent object class in the object class hierarchy. The DIT superior object classes indicate under which entries an entry of the given object class can appear. The SUP object class and the DIT superior object classes describe data modeling constraints in the object class hierarchy and DIT structure, respectively.
As described previously, the object classes defined for a particular purpose should be grouped into a package that can offer some version control over the schema definition itself. The object class names for the data model should be contained in the package definition.

Every attribute type and object class defined in the directory schema must have a unique identifier. Data modelers must be free to assign new identifiers without contacting a central authority, yet the identifiers must be unique so that different users do not misinterpret information in the directory service. An OID format was defined in the X.500 data model to cover this. An OID is a string of integers of arbitrary length. One way to view the string of integers is as a tree. If an OID has only two integers (for example, 2.4) then the OID is near the top of the tree. Longer strings represent lower arcs in the tree (for example, 1.3.18.0.2). If different organizations are assigned different arcs in the tree and are constrained to defining OIDs only under that arc of the tree, then the definition of OIDs can be distributed across the world and still allow unique OID definitions. This is how OIDs are administered. Organizations obtain an arc in the OID tree from a higher authority, then use that arc to define new OIDs for object classes and attribute types.

The OID setting is required for object classes and attribute types defined in the schema. These should be assigned using an arc in the OID tree that is assigned to the organization creating the data model.

**Defining Attribute Types**

Next the attribute types are introduced. The definition of an attribute type includes a specification of the attribute type’s syntax, matching rule(s), OID, and whether the attribute type is single-valued or multivalued.

Attribute types define information content in the directory service. So that multiple applications can interpret this data correctly, the format of the information must be defined. The format of an attribute value for an attribute type is defined by the syntax for the attribute type. The syntax must be specified for new attribute types that are defined.

Related to syntax is the set of matching rules that are to be used with the attribute type. Matching rules determine how attribute type values will be treated when searching the directory service for information. For example, the syntax for an attribute type might be defined as IA5String, indicating that attribute values for the attribute type will consist of a string of 7-bit ASCII characters. For this attribute type, the equality matching rule might be specified as caseIgnoreIA5Match. If so, upper- and lowercase letters will be treated the same in search operations. Alternatively, if caseExactIA5Match is used, upper- and lowercase letters would be treated as different. In addition to equality
matching, a substrings matching rule can be specified. Like the equality matching rule, the substrings matching rule is related to the syntax of the attribute type.

Object class and attribute type definitions for a data model should be gathered into a package. Together, the attribute types and object classes will form a package that can be identified, can be referred to, and can contain version information as well as a description.

Attribute types must also have an OID assigned. Just like object classes, the OID ensures that the attribute type definition is unique. The attribute type definitions for our example appear in Code Example 17.3.

Now that all of the object classes and attribute types for the new schema have been defined, the contents of the package can be defined. Code Example 17.4 shows the package contents for our example.

In Code Example 17.4, the definition of the directory entry that represents the package is shown in LDAP data interchange format (LDIF). (Refer to Appendix H for a pointer to information about this interchange format.) In LDIF format, each line represents an attribute value with the attribute type provided before a colon (:) separator and the attribute value provided after the colon. The first line of the LDIF entry definition indicates the distinguished name (DN) of the entry.

<table>
<thead>
<tr>
<th>Attribute Type Definitions for the Attribute Types in the Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1.3.18.0.2.1000.1.1.17.4.4 NAME 'companySerNo' SINGLE-VALUE EQUALITY 'integerMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 )</td>
</tr>
<tr>
<td>( 1.3.18.0.2.1000.1.1.17.4.5 NAME 'groupNames' EQUALITY 'distinguishedNameMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12 )</td>
</tr>
<tr>
<td>( 1.3.18.0.2.1000.1.1.17.4.6 NAME 'groupLeader' EQUALITY 'distinguishedNameMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12 )</td>
</tr>
<tr>
<td>( 1.3.18.0.2.1000.1.1.17.4.7 NAME 'managerName' EQUALITY 'distinguishedNameMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12 )</td>
</tr>
<tr>
<td>( 1.3.18.0.2.1000.1.1.17.4.8 NAME 'affiliations' EQUALITY 'caseIgnoreMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )</td>
</tr>
<tr>
<td>( 1.3.18.0.2.1000.1.1.17.4.9 NAME 'internalPhone' EQUALITY 'telephoneNumberMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.50 )</td>
</tr>
<tr>
<td>( 1.3.18.0.2.1000.1.1.17.4.10 NAME 'internalAddress' EQUALITY 'caseIgnoreMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )</td>
</tr>
</tbody>
</table>
dn: packageName=aCorporateSchema, cn=installedSchemas, o=yourCompany
objectclass: package
packageName: aCorporateSchema
version: 1.0
description: Object classes and attribute types representing persons, groups, and departments in the a organization
objectclasses: ( 1.3.18.0.2.1000.1.1.17.6.2 NAME 'aPerson'
  STRUCTURAL SUP 'person' MUST 'companySerNo' MAY 'groupNames'
)
objectclasses: ( 1.3.18.0.2.1000.1.1.17.6.3 NAME 'aGroup'
  ABSTRACT SUP 'Group' MUST ( 'companySerNo' $ 'groupLeader' )
)
objectclasses: ( 1.3.18.0.2.1000.1.1.17.6.4 NAME 'aDepartment'
  STRUCTURAL SUP 'organizationalUnit' MUST 'managerName'
)
objectclasses: ( 1.3.18.0.2.1000.1.1.17.6.5 NAME 'aManager'
  STRUCTURAL SUP 'aPerson'
)
objectclasses: ( 1.3.18.0.2.1000.1.1.17.6.6 NAME 'aBoardMember'
  STRUCTURAL SUP 'aPerson' MAY 'affiliations'
)
objectclasses: ( 1.3.18.0.2.1000.1.1.17.6.7 NAME 'aDistGroup'
  STRUCTURAL SUP 'aGroup'
)
objectclasses: ( 1.3.18.0.2.1000.1.1.17.6.8 NAME 'aAccessGroup'
  STRUCTURAL SUP 'aGroup'
)
objectclasses: ( 1.3.18.0.2.1000.1.1.17.6.9 NAME 'aContactInfo'
  AUXILIARY SUP 'top'
  MUST ( 'internalAddress' $ 'internalPhone' )
)
attributeTypes: ( 1.3.18.0.2.1000.1.1.17.4.4 NAME 'companySerNo'
  SINGLE-VALUE EQUALITY 'integerMatch' SYNTAX
  1.3.6.1.4.1.1466.115.121.1.27
)
attributeTypes: ( 1.3.18.0.2.1000.1.1.17.4.5 NAME 'groupNames'
  EQUALITY 'distinguishedNameMatch'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.12
)
attributeTypes: ( 1.3.18.0.2.1000.1.1.17.4.6 NAME 'groupLeader'
  EQUALITY 'distinguishedNameMatch'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.12
)
attributeTypes: ( 1.3.18.0.2.1000.1.1.17.4.7 NAME 'managerName'
  EQUALITY 'distinguishedNameMatch'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.12
)
attributeTypes: ( 1.3.18.0.2.1000.1.1.17.4.8 NAME 'affiliations'
  EQUALITY 'caseIgnoreMatch'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
)
attributeTypes: ( 1.3.18.0.2.1000.1.1.17.4.9 NAME 'internalPhone'
  EQUALITY 'telephoneNumberMatch'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.50
)
attributeTypes: ( 1.3.18.0.2.1000.1.1.17.4.10 NAME 'internalAddress'
  EQUALITY 'caseIgnoreMatch'
  SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
)

Code Example Q.4  A package describing the new schema
Creating DN-Pointers

Attribute types that are defined to hold the names of other entries in the directory name space should be defined with distinguishedName syntax. This syntax has a matching rule associated with it, distinguishedNameMatch, that takes into account the characteristics of each attribute type in the distinguished name when performing matching operations.

Because these attribute types will contain the dn-pointers to other entries in the directory, special care must be taken to maintain referential integrity. In the example object classes, the groupNames attribute type is a multivalued dn-pointer that is defined as part of the aPerson object class. When a new person is added to a group, the entry representing the person should be updated with an attribute value stored in the groupNames attribute that points to the aGroup object in the directory name space.

Adding a user to a group, in this case, requires two operations: adding an attribute value to the aGroup entry and adding an attribute value to the aPerson entry. Since the directory service defines atomicity characteristics at the individual entry level, special care should be taken to clean up if one of the modify operations fails. In the case that error recovery fails to clean up the partial update of the directory, the application should provide some indication of what manual cleanup is required.

Because group membership is such a common construct, a number of directory implementations support referential integrity characteristics for group entries. These characteristics are usually constrained to the object classes and attribute types that the directory implementation understands, and are not available for attribute types and object classes defined by new applications.

Defining Functional Characteristics

The static definition of object classes and attribute types is rarely sufficient for describing how to use the newly defined schema. In addition to the static definition, a description of the dynamic characteristics for the object classes and attribute types should be provided. This description should include any special processing that must be performed as part of operating with entries that conform to the schema definitions. Examples of this special processing include managing dn-pointers and generating unique values to store in the entries that are being created.

In the example schema there is a special process for handling aDepartment entries in setting up the dn-pointer to the manager of the department. It is assumed that an entry, defined as a name, can be contacted whenever the manager for a department is not yet defined. This name should be used if the manager has changed positions or is no longer part of the organization. Thus, when updates are made to the aPerson entry that repre-
sents the manager of a department, the aDepartment entry might also require updates. When the manager leaves the organization, the aDepartment entry should be updated to list the default contact entry stored in the directory. (See Figure 17.4.)

Other types of special processing may be necessary as well. All of this information should be described as part of the schema definition and accompany the definition of the schema used by an organization or application. Be sure to include these semantics in a description of the schema. The description attribute in the package object class is a good place to store this information.

**Business Issues Impacting Structure**

When defining a new schema, pay attention to the structure of the new entries in the DIT. Several business issues, such as access control, required isolation of subsystems, and required separation between departments can affect the structure of the directory name space.
A “deeper” hierarchy of entries (a hierarchy where there are fewer sibling entries and more descendant levels) can be used to support drill-down capabilities. This lets you remove more specific or specialized information from the upper layers of the hierarchy and save it for detailed administration or analysis. Keeping the upper layers of the directory name space more general will make the directory service easier to use.

A deeper hierarchy can also support the need for isolation between departments. Coupled with granular access controls placed on entries in the directory service, a deeper hierarchy can hide information that should not be shared in lower layers of the directory name space.

A “shallow” hierarchy of entries (a hierarchy in which there are many siblings and fewer descendant levels) can be useful in some environments. If information is scattered, deep in the hierarchy, it can be hard to find without performing very general searches of the directory service. For some applications, a “flatter” hierarchy provides a single entry to look under, which can be simpler for some applications.

Let the business problem to be solved guide your choice of whether to build a deep or shallow hierarchy of entries.

### Using the Base Schema

When defining a new schema, or any time for that matter, make use of existing work where it makes sense to do so. A good place to start when defining a new schema is the set of object classes and attribute types that are defined in the X.520 standard, which is part of the X.500 family of standards, and IETF RFC 2252, which is part of the set of IETF RFCs that make up LDAP Version 3. If new object classes and attribute types will be defined, consider using the classes and types in X.520 and RFC 2252 as a basis for the new definitions. Refer to Appendix I for descriptions of attribute types and object classes to use as a starting point for defining new schemas.

Another alternative is to use the IBM schema as a basis for the new schema. The IBM schema is described in Appendix G. The IBM schema merges the concepts in the attribute types and object classes in X.520 and RFC 2252 with the CIM model and work from the DMTF on interoperable directory schema. Refer to Appendix H for copies of RFC 2252 and other RFCs that are related to RFC 2252.

### Installing a Custom Schema

Once a schema has been defined, the directory server that is providing the directory service must be updated with the new schema definitions so they can be used by the application. With LDAP Version 3, there is a defined mechanism for using the LDAP protocol itself to update schema information in the directory server. The
LDAP Version 3 specification defined a subschema object class so that directory servers could describe the attribute types and object classes available for use.

An entry in the directory name space called the subschema subentry contains the current set of attribute types and object classes that can be used to store information in the directory server.

The IBM SecureWay Directory Server supports updates to the subschema subentry in the directory name space. The newly defined attribute types and object classes can be added to the schema in the directory server to allow the object classes and attribute types to be used by applications. It is important to note that the operation to add the schema information to the directory server is an LDAP modify operation, not an add operation. Keep in mind that the directory server already contains schema information to cover the object classes and attribute types already contained in the directory information tree. Also, these object classes and attribute types are represented as attribute values of the two multivalued attribute types: objectClasses and attributeTypes. Thus, adding new object classes and attribute types is really a modification of the objectClasses and attributeTypes attributes in the subschema subentry held by the directory server.

Code Example 16.1 showed how to determine the distinguished name of the subschema subentry for the directory server. Once this name is known, it can be used to update the schema information.

**Updating a Schema Using LDAP Version 3**

Code Example 17.5 shows how to take a new schema definition and add it into the directory server.

**Updating a Schema Using LDIF Format**

Instead of writing a program to update the schema information, you can contain the new schema information in an LDIF file and use a tool that understands LDIF to update the schema.

With some LDAP installations, a command-line tool called `ldapmodify` is available for these types of operations. Code Example 17.6 shows how an LDIF file can be used to define the new schema; Code Example 17.7 shows how the `ldapmodify` command is used to process this LDIF file and make the updates to the schema used by the directory server.
char *attrTypeVals[] = {
    "( 1.3.18.0.2.1000.1.1.17.6.2 NAME 'aPerson' STRUCTURAL SUP 'person' MUST 'companySerNo' MAY 'groupNames' )",
    "( 1.3.18.0.2.1000.1.1.17.6.3 NAME 'aGroup' ABSTRACT SUP 'Group' MUST ( 'companySerNo' $ 'groupLeader' ) )",
    "( 1.3.18.0.2.1000.1.1.17.6.4 NAME 'aDepartment' STRUCTURAL SUP 'organizationalUnit' MUST 'managerName' )",
    "( 1.3.18.0.2.1000.1.1.17.6.5 NAME 'aManager' STRUCTURAL SUP 'aPerson' )",
    "( 1.3.18.0.2.1000.1.1.17.6.6 NAME 'aBoardMember' STRUCTURAL SUP 'aPerson' MAY 'affiliations' )",
    "( 1.3.18.0.2.1000.1.1.17.6.7 NAME 'aDistGroup' STRUCTURAL SUP 'aGroup' )",
    "( 1.3.18.0.2.1000.1.1.17.6.8 NAME 'aAccessGroup' STRUCTURAL SUP 'aGroup' )",
    "( 1.3.18.0.2.1000.1.1.17.6.9 NAME 'aContactInfo' AUXILIARY SUP 'top' MUST ( 'internalAddress' $ 'internalPhone' ) )",
    NULL },

char *objClassVals[] = {
    "( 1.3.18.0.2.1000.1.1.17.4.4 NAME 'companySerNo' SINGLE-VALUE EQUALITY 'integerMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.27 )",
    "( 1.3.18.0.2.1000.1.1.17.4.5 NAME 'groupNames' EQUALITY 'distinguishedNameMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12 )",
    "( 1.3.18.0.2.1000.1.1.17.4.6 NAME 'groupLeader' EQUALITY 'distinguishedNameMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12 )",
    "( 1.3.18.0.2.1000.1.1.17.4.7 NAME 'managerName' EQUALITY 'distinguishedNameMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12 )",
    "( 1.3.18.0.2.1000.1.1.17.4.8 NAME 'affiliations' EQUALITY 'caseIgnoreMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.12 )",
    "( 1.3.18.0.2.1000.1.1.17.4.9 NAME 'internalPhone' EQUALITY 'telephoneNumberMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.50 )",
    "( 1.3.18.0.2.1000.1.1.17.4.10 NAME 'internalAddress' EQUALITY 'caseIgnoreMatch' SYNTAX 1.3.6.1.4.1.1466.115.121.1.15 )",
    NULL },

LDAPMod mod1 = { LDAP_MOD_ADD, "attributeTypes", NULL };
LDAPMod mod2 = { LDAP_MOD_ADD, "objectClasses", NULL };
LDAPMod *mods[] = { NULL, NULL, NULL };
char *subSchemaEntryDN = "cn=schema";
mod1.mod_values = attrTypeVals;
mod2.mod_values = objClassVals;
mods[0] = mod1;
mods[1] = mod2;
rc = ldap_modify_s( ld, subSchemaEntryDN, mods );

Code Example Q.5  Adding object classes and attribute types to the directory server's schema
dn: cn=schema
changeType: modify
add: x
objectclasses: { aPerson OID 1.3.18.0.2.1000.1.1.17.6.2
    STRUCTURAL SUP Person MUST companySerNo MAY groupNames }
objectclasses: { aGroup OID 1.3.18.0.2.1000.1.1.17.6.3
    ABSTRACT SUP Group MUST companySerNo $ groupLeader }
objectclasses: { aDepartment OID 1.3.18.0.2.1000.1.1.17.6.4
    STRUCTURAL SUP organizationalUnit MUST managerName }
objectclasses: { aManager OID 1.3.18.0.2.1000.1.1.17.6.5
    STRUCTURAL SUP aPerson }
objectclasses: { aBoardMember OID 1.3.18.0.2.1000.1.1.17.6.6
    STRUCTURAL SUP aPerson MAY affiliations }
objectclasses: { aDistGroup OID 1.3.18.0.2.1000.1.1.17.6.7
    STRUCTURAL SUP aGroup }
objectclasses: { aAccessGroup OID 1.3.18.0.2.1000.1.1.17.6.8
    STRUCTURAL SUP aGroup }
objectclasses: { aContactInfo OID 1.3.18.0.2.1000.1.1.17.6.9
    AUXILIARY SUP top MUST internalAddress $ internalPhone }
attributeTypes: { companySerNo OID 1.3.18.0.2.1000.1.1.17.4.4
    SINGLE-VALUE SYNTAX Integer }
attributeTypes: { groupNames OID 1.3.18.0.2.1000.1.1.17.4.5
    SYNTAX DistinguishedName }
attributeTypes: { groupLeader OID 1.3.18.0.2.1000.1.1.17.4.6
    SYNTAX DistinguishedName }
attributeTypes: { managerName OID 1.3.18.0.2.1000.1.1.17.4.7
    SYNTAX DistinguishedName }
attributeTypes: { affiliations OID 1.3.18.0.2.1000.1.1.17.4.8
    SYNTAX DirectoryString }
attributeTypes: { internalPhone OID 1.3.18.0.2.1000.1.1.17.4.9
    SYNTAX telephoneNumber }
attributeTypes: { internalAddress OID 1.3.18.0.2.1000.1.1.17.4.10
    SYNTAX DirectoryString }

Code Example Q.6  LDIF file containing updates to the schema

$ Ldapmodify -h localhost \
-D <bindDN> -pw <password> \ 
-f <ldiffFileName>

Code Example Q.7  Using Ldapmodify to update the schema with an input LDIF file
Other Schema Definition Formats

There are additional formats for defining directory service schema. Emerging work on the directory service markup language (DSML) will allow directory schema to be described using the extensible markup language (XML). Using XML will ensure that schema information can be exchanged between interested parties in an easily parsed format.

To keep track of the complex relationships between attribute types, object classes, and packages, a relational database can be used to verify the referential integrity of the schema itself prior to deploying the schema on the directory server. Simple database reports can be produced that create LDIF files, which can be used to update the schema in the directory server.

Summary

This chapter presents the steps to take when building a schema customized for an application or organization. The extensibility of the directory service allows it to be applied to solve many problems in publishing and finding information in a distributed processing environment. Building schemas will become a common task in exploiting the capabilities of the directory service across the enterprise.

To define a schema, first model the information to be stored in the directory. Next, define the attributes for each entity and the relationships between entities. Use the hierarchical structure of the directory name space to represent relationships and dn-pointer attributes to represent other relationships. Using the data model define attribute types and object classes. Define a package to represent the set of attribute types and object classes in the schema. Finally update the schema defined in the directory server to include the new attribute types and object classes.
Chapter 18
DIRECTORY SECURITY

“Money couldn’t buy you friends, but you got a better class of enemy.”
—SPIKE MILLIGAN

This chapter covers the main security services of authentication, access control, auditing, data confidentiality, data integrity, and security management. These services are the building blocks for implementing an enterprise’s directory security policy. They need to be applied in conjunction with the enterprise application design in order to meet the needed security requirements.

User Identification and Authentication

Enterprise directories can be used to consolidate user definitions. Users definitions in a directory can be accessed by multiple applications, allowing a single user definition to be applied across the applications. In addition to authentication credentials, the directory can hold user personalization and configuration information. Organization units and other directory containers can act as administration control points, improving separation of administrator authority. The combination of directory servers and replication can increase both availability and performance.

The LDAP API has several BIND operations that can be used to perform user authentication. Each type of BIND provides for a different set of authentication options.

* ldap_simple_bind(_s) — Simple BIND authenticates using a DN and clear text password.
* ldap_bind(_s) — Regular BIND allows the specification of the user credentials and the type of authentication to be used. Starting with LDAP Version 3, the use of ldap_sasl_bind is recommended instead of ldap_bind.
* ldap_kerberos_bind(_s) — The Kerberos BIND requests that Kerberos authentication be used. Most directories do not implement this function. ldap_sasl_bind should be used instead.
* ldap_sasl_bind(_s) — SASL BIND uses the Simple Authentication and Security Layer (SASL) to specify and negotiate the security used by the directory connection.
The various forms of BIND allow the program to pass the user’s authentication credentials to the directory. \texttt{ldap\_simple\_bind()} handles the simple cases of anonymous access and authenticating with a password. However, if any user authentication is required, \texttt{ldap\_bind()} and \texttt{ldap\_sasl\_bind()} are better choices. Both of these functions support authentication methods that do not expose user passwords in clear text.

Instead of binding using a user’s ID and password, an alternative authentication method is to bind using a service’s ID and have the server search for the user’s directory object and compare the password. This method has performance advantages because a single ldap connection can be re-used to validate many users. It has the disadvantage of restricting the authentication methods that can be used. The user must send a clear text password to the server. Additionally, the directory cannot use the user’s identity to perform access control since the directory did not directly authenticate the user. This also requires that the server be given compare access to the user’s password attribute in the directory.\footnote{Issuing a compare against a user’s password may not be allowed, depending on the security policy.}

The middle-tier application must decide whose identity to use in accessing the directory and other resources: the originating user’s identity or the server’s identity. This leads to a number of trade-offs related to performance, scalability, and security.

Figure 18.1 shows a typical configuration with a Web application accessing several backend systems. Whose identity should be used to access the various backend systems? Let’s consider the types of data and related security policies of the backend systems.

- Directory—contains customer authentication data and customer preferences. Customer data is confidential and must not be disclosed to other customers.
- Catalog database—contains the company’s catalog, including inventory information. Most of this data is not confidential. The middle-tier server modifies none of the data.
- Credit card processing system—third-party system used to charge customer credit cards for orders. This system requires a user ID assigned to the company.
- Order transaction system—records orders in the core business systems. Order data is considered confidential. In addition, because the credit card processing and order transaction systems are separate, the ability to enter orders bypassing the credit card processing system can be used to commit fraud.
Now which IDs should be used? The customer’s identity can be used to access the directory. This will help ensure that one customer cannot, accidentally or intentionally, access another customer’s data. Access to the catalog database should be done under the server’s identity. The catalog database does not contain much confidential information. There is limited value in controlling the access at the database itself. The credit card processing system requires a specific ID, and this ID must be used. There is some danger with the credit card processing design. Because the ID and password are stored on the middle tier, any successful intrusion could expose this valuable ID and password. A better structure would be to place an intervening server between the credit card processing system and the middle tier, perhaps even to integrate the credit card charging functions into the order transaction system. The order transaction system is a harder choice. While using the customer’s ID would help with auditing, it means that external users would be granted privileges to a core business process. Using a server’s ID greatly reduces the number of IDs that require order transaction processing privileges. The auditing concerns can be addressed by having the server provide the originating user’s identity, after it has been authenticated.

An additional method worth describing is using some form of user identity mapping. If a backend system does not support the user IDs that are used by the directory, the directory can be used to create and implement ID-mapping facilities. In this case, before accessing the backend system, the server would access the customer’s directory entry to retrieve a different ID to be used to access the backend system. This mapping does not have to be one-to-one; it is possible to map multiple directory

---

**Figure R.1** Common Web service with multiple backend systems
users to the same backend identity. This allows role-based access control, by mapping sets of users to role-related IDs.

**Controlling Access to Directory Entries**

While the general intent of the first directories was to provide read access to everyone, current usage includes the need to protect the confidentiality of some directory data. Additionally, even from the beginning there was a need to protect directory data from unauthorized modification. A current challenge with directory access control is the lack of standardization.

Directory access control is used in two fashions: to protect objects that reside within the directory and to use proxy objects in the directory to control access to nondirectory resources. Both usages require that a directory support some form of access control lists. In general, programs do not have to be aware of how access control lists are formed or checked. The main exception is when they create new objects that need nondefault protection.

Let’s briefly overview the access control model of Active Directory. Appendix F provides a longer description of the Active Directory access control model, as well as describing the IBM SecureWay Directory and Netscape Directory servers access control models.

Active Directory’s access control lists are an extension of Windows NT access control lists. Access control lists (ACLs) consist of a header and an ordered list of access control entries (ACEs). Each ACE has a type (allow or deny), a mask of permissions, some flags, and a security identifier (SID) representing the user or group that the entry applies to. Active Directory adds new types (allow_object and deny_object) that include two new ACE fields, object type and inherited object type. Figure 18.2 shows how an Active Directory security descriptor and ACL entries are structured.

The allow_object and deny_object ACE type includes a schemaGUID (the ObjectGuid field) that modifies what the ACE applies to. The schemaGUID matches a GUID that defines one of a class, an attribute, or a property set. This restricts the ACE to only being involved in access control decisions related to the specific class, attribute, or attributes in a property set.

The basic access evaluation algorithm is unchanged from Windows NT. This algorithm is to scan the ACL (looking at ACEs in order) until either one requested permission is denied or all requested permissions are accumulated.

How can directories be used to protect a nondirectory object? While it is obvious that directory access control can be used to protect directory objects, using the directory to protect nondirectory objects takes more creativity. This requires the defini-
tion of proxy or shadow objects in the directory that can be used as surrogates for the real objects when checking access control. (Code sample ldapauth in Chapter 12 is one example of how this might be implemented.) Care must be taken to ensure that the surrogate objects actually reflect the real objects. This also does not work well if the objects are created and deleted frequently, since directories are optimized for more stable object existence.

While surrogate objects work well for controlling read access, they are not as useful for controlling write access. Checking for read access can be done using the direct approach of attempting to retrieve the object with `ldap_search`. If the object has multiple types of read access, attempting to read a specific attribute for each type of read access can control this. Write checking can be done in basically the same fashion,
using `ldap_modify` instead of `ldap_search`, but this means determining which attribute to attempt to modify.

A big problem with using a directory in this fashion is the performance overhead of requiring an `ldap_bind` and `ldap_search` or `ldap_modify` for each user access. This overhead is likely to be substantial when the directory does not reside on the server’s system. However, the ability to use a single database (directory) for access control decisions from multiple servers and services can greatly reduce the cost of administering security while improving access consistency.

### Data Confidentiality and Data Integrity

Data confidentiality and data integrity deal with the need to protect data from disclosure or modification while it is being transmitted across a network. Directory protocols provide for this service as an integral part of the directory protocol definition. Lotus Notes provides integrated encryption usage that can protect all data sent by a Notes client to any Domino server. LDAP provides for the usage of secured sockets layer (SSL) to protect data sent between an LDAP client and an LDAP-based server. (See Figure 18.3.)

SSL encrypts the data in transit using a secret key and the RC4 algorithm. The secret key is generated in a protected exchange between the client and the server. An LDAP server determines whether SSL should be used, based on the socket that the request arrives at and the first few bytes of received data. This allows requests for

![Figure R.3](image)

**Figure R.3**  SSL can be used to protect LDAP exchanges.

2. RC4 is not the only algorithm that can be used. Others include DES, IDEA, and Triple DES.
data that do not require confidentiality or integrity protection to be made without paying the overhead cost associated with encrypting the data.

An additional confidentiality technique is to encrypt the data before it is stored in the directory. This has the advantage of ensuring that the data is protected, even if an attacker can subvert the directory service. However, it has the major disadvantage of requiring the using application to manage encryption keys and can greatly limit the ability of the data to be shared across multiple applications. Similarly, data integrity can also be achieved by storing cryptographic checksums in the directory. The application can, by recomputing the checksum and comparing it to the stored value, verify that the data is unchanged. Of course, this requires that the application trust the integrity of the checksum value.

Security Management and Administration

The deployment of enterprise directories will almost certainly create a number of organizational problems. As directory usage increases, more data will be moved into it. The organizations that currently own and manage the data elements may not want to give up administrative control. At the same time, other organizations that could use the data are likely to resist being dependent on data under another group’s control. This is especially true of the desire to control the security administration of critical data.

The current security policy will need to be applied to the directory data. This will require implementing security procedures to ensure that the policy elements are correctly implemented. It will also demand that business areas realize that shared data have (in general) more value.

Summary

A directory both raises new security concerns and can be used to fix existing security problems. They can be used to provide authentication services for users and middle tier servers. Directory access control facilities are useable in centralizing access decisions. The LDAP protocol provides confidentiality and integrity services using SSL. While the security models of most directories are still maturing, the consolidation and control point they provide brings significant security benefits. The administrative cost reductions alone will ensure that directories continue to be deployed.
Chapter 19

REPLICATION AND PARTITIONING

Replication and partitioning are the yin and yang of directory infrastructure. (See Figure 19.1.) Separately, they are powerful and necessary tools for maintaining the data contained in a directory. Taken together, replication and partitioning enable directory data to achieve enterprise characteristics.

The fundamental ideas are very simple. Replication means exactly what it sounds like: making multiple copies of data (not just simple caches for performance). Replicas of a directory are copies of the directory’s data stored in a different server (at least a different software server, but likely a different hardware server as well). Partitioning is only slightly more complicated. Partitioning is the act of carving a subset or subtree from a larger tree data, probably to relocate it physically.

Replication and partitioning are used to achieve the same and different results. Replicas can be used to increase data availability, or to increase performance characteristics by spreading out the server load. Partitioning can create smaller units to administer or increase performance by locating the data closer to clients or by creating smaller databases to facilitate search operations.
Replication

Replication of data can be between like directory servers (from the same vendor, for example) or between dissimilar directory servers. We will focus on replication between like servers and call it synchronization when it is between dissimilar directory servers.

In replication, the details are usually hidden inside the implementation. Synchronization between dissimilar directory servers is often either a manual process or requires third-party tools, such as metadirectories or other synchronizing software (see Chapter 20 and Chapter 29 for metadirectories and synchronization). (See Figure 19.2.)

How Replication Works

The process of replicating directory data ranges from “write your own synchronization agents” to “let the directory do it for you.” Writing your own synchronization agents—software that makes appropriate, simultaneous changes to data in multiple locations—is obviously a last resort: Synchronization agents are more difficult to get right, and generalized, than they might at first seem. (See Figure 19.3.)

Directory servers usually come ready to replicate themselves. That is, through administrative action, you tell a directory server on one machine that it is a replica of a directory server on another machine. The directory servers then cooperate by sending each other any changes that occur. Ideally, these directory servers could be from different vendors, but practically speaking that does not yet occur. IETF LDAP replication standards may change this in the future (see Appendix H for links to IETF standards and drafts).

Figure S.2  Replicating and synchronizing data between like and dissimilar directory servers
There are several schemes that directory servers might use to coordinate replication between themselves. The two primary categorizations are single master and multimaster. In a single master scheme (see Figure 19.4), one directory server is anointed the king, or master. The master server is the one on which changes can occur and then it propagates the changes to the other subordinate servers. In a multimaster scheme (see Figure 19.5), each server is a read-write copy that can be changed. When changes occur in a multimaster directory scheme, it is more complicated for the directory servers to coordinate their updates. Comparing Figure 19.4 with Figure 19.5, you can see why multimaster operations are harder to implement.

Figure S.3  A secret synchronization agent (a secret from the directory server)

There are several schemes that directory servers might use to coordinate replication between themselves. The two primary categorizations are single master and multimaster. In a single master scheme (see Figure 19.4), one directory server is anointed the king, or master. The master server is the one on which changes can occur and then it propagates the changes to the other subordinate servers. In a multimaster scheme (see Figure 19.5), each server is a read-write copy that can be changed. When changes occur in a multimaster directory scheme, it is more complicated for the directory servers to coordinate their updates. Comparing Figure 19.4 with Figure 19.5, you can see why multimaster operations are harder to implement.

Figure S.4  Single master operation
Application-Level Problems Caused by Replication

Replication causes some problems for application writers. One problem is that when data change in one replica, it can take some time before the data changes in replicas. This change latency is one cause of loose consistency (see Chapter 2). Directory data are loosely consistent, although over time they converge.

Latency Issues and Loose or Temporal Consistency  How long it takes for a change to be reflected in all of the replicas depends on the speed of the network between the replicas, the algorithm used by the directory service for replication propagation, and administrative settings. (See Figure 19.6.)

Replica propagation can be implemented a variety of ways. To save on network bandwidth, changes can be batched and sent only every 15 minutes, for example, or changes may be replicated immediately. A directory service can implement a range of replication frequency options for a range of data classifications (maybe from don’t-care, for batched updates through replicate-immediately for data that must be synchronized as soon as possible after a change).

Replication Versus Caching

Replication is not the same as simple caching. Often a cache is just a set of frequently used information kept locally to improve performance. A replica is more compli-
cated. While the information in a cache is read-only and may or may not be up to date, the information in a replica is “real”—it may be read-write and is loosely consistent with the data in other replicas.

Dealing with Temporal Consistency

The following program demonstrates a technique for dealing with data that are changing in the directory while you are using them. The program retrieves connection information for a server and contacts the server. Instead of giving up the ghost if the server can’t be contacted, the program goes back to the directory to check whether the address is still valid. Variations on this theme abound. For example, the program might perform this directory retry a given number of times before alerting a human of the problem, or it might even keep running and use cached information, going back for newer connection information after some specified amount of time.

This sample program is called Retry. The Retry program is a client that looks for services in a directory. It finds all of the servers of a particular type that have advertised themselves in the directory. Retry then attempts to talk to one of the servers it has found, using binding information in the server advertisement. Each server put its IP and port number in the directory so clients can contact it—this is an old concept that illustrates techniques for dealing with temporal consistency and stale data.

If Retry is able to connect with a chosen server, it communicates with it and that’s pretty much the end of the story. The more interesting case is when Retry attempts to contact a chosen server but fails. The failure could be caused by any number of things, chiefly that the server may have crashed hard and was unable to remove its
advertisement from the directory. What can you do with data that may be stale, or just plain wrong? Retry allows the user to choose among the following.

- Attempt to contact the same server again. Maybe it came back up, or there was just a temporary network glitch that prevented the first connection attempt from succeeding.
- Choose another server from among the ones found during the directory search. If one server doesn’t work, try another one that exports the same function.
- Refetch service advertisements from the directory and start over. This is “last resort” logic. It could be that all of the data were bad. The program tries to get them again, hoping that they changed (replication updated them, or the server we are trying to contact came back up on a different port and readvertised).

The Retry sample depends on finding a server that advertises itself, so we’ve re-used the server example from Chapter 9 for this purpose. An easy way to test the Retry program is to start the server program several times, so that there are several advertised instances to be found in the directory. “Crash” a few of them so that there are advertisements in the directory that are no longer valid.

Figure 19.7 shows the Retry client after it searched the directory, found three advertisements, chose one of them, and successfully connected to a server at IP address 192.168.0.1 and port 1729. Figure 19.8 shows the retry client after it searched the directory, found three service advertisements, chose one of them, and failed to connect to the server at IP 192.168.0.1 and port 1721. We don’t know why it failed, only that the socket connection failed with a return of WSACONNREFUSED (a Windows Sockets, or Winsock, return indicating that “no one was home” at that port).
Figure 19.8 shows that when Retry discovered the connection failure, it asked the user to choose among trying again, trying another server from the list previously read from the directory, and refetching the list of advertisements from the directory and starting over. The user decided to pick another server from the list and the Retry program is waiting for the user to choose the specific one.

**Replication Collisions** Let’s say that you and your sister both have a recipe that was copied and given to you by your mother. You both make extensive changes to the recipe and you both give it to a mutual friend. As far as your friend can tell, it looks like two different recipes. This happened because you and your sister behaved like typical siblings—that is, you didn’t share. Your friend is a victim of a change collision, because two copies of something underwent simultaneous, unreconciled changes. As for your friend, well, let’s hope that you and your sister are both good cooks.

A similar problem exists in a multimaster directory replication scheme, where a change occurs to two (or more) copies of the same object (see Figure 19.9). Someone has to reconcile the changes. Programs frequently make changes so a directory that notices a change collision could find one of the two (or more) agents of change and say, “oops, that change you made fifteen minutes ago can no longer be honored.” That is especially difficult if the program that made the change has exited.
We can fix this problem by making a program wait until all replicas have received and updated the change only if we lock the object being changed and make sure that no one else changes it while we are changing it, on all of the replicas. This is what databases do and they pay for it dearly (see Chapter 2 and Chapter 14).

Another way to deal with change collisions is to make sure they don’t happen yourself (as opposed to expecting a directory to do it for you) or to make sure that the collisions don’t matter. Here are some ways to do those things.

- Make sure only one process can update the data. Let all other processes read it; they might still get stale data but it won’t be data that is colliding with other data.
- Write your own locking mechanism. This is similar to a distinguished write process but is more generalized (and a lot harder to write).
- Write programs that realize the data they are reading out of the directory may be stale or even wrong. For example, if the program retrieves the address of a server and finds it isn’t really at that location, it can retrieve the address again. If the address isn’t the same the second time, the data was probably stale in the process of reconciliation.
- Use read-only data, or effectively read-only data. This is the easiest way of all. Effectively-read-only data are data that only change “once in a blue moon.”

**More Complications of Replication Collison** The Retry sample is a simple, useful demonstration of dealing with potentially stale data. It didn’t really matter that the data came out of a directory. If you advertised a service in a database, the advertisement could still be stale. That is, an advertised server might have gone down hard and failed to update its advertisement data, no matter where it was stored. That’s just the nature of advertisement data; a database doesn’t buy you any more than a directory in this case.

With stale data you can simply go back and fetch another round and see if those data are any better (the server may have come back up and readvertised, for example). A
potentially more serious problem concerns pieces of data that need to be in a consistent state relative to each other. Most directories are not equipped to handle this well. For example, if you update two attributes on the same directory entry, LDAP Version 3 says that the change is atomic, “at the originating server.” Does that mean there are atomic updates to one entry or not? The answer is both yes and no.

Perhaps two attributes must be consistent. Say one attribute is a simple boolean that indicates whether someone is an executive or not, and the other attribute can take on certain values only if the person is an executive (and it must not have those values otherwise—like “executive stock options”). When someone is promoted to executive, your program can update both the executive attribute and the options attribute with “executive stock options” in one atomic operation, “at the originating server.”

If one executive user object exists in two directory replicas, and the replicas are both fully read-write (a multimaster replication scheme), what happens if the two user objects are changed at roughly the same time? One change updates the executive attribute to true and adds executive stock options (an atomic update to that one user entry at that server). The other change, to the other replica, adds a type of options to the user entry that executives are not allowed to have (this is okay, since the user in this replica is not an executive—yet). It is possible, though unlikely, that through replication we can end up with a user entry that is both executive and has a type of options that executives are not allowed to have. It isn’t likely for several reasons, chiefly timing and the fact that it is a very odd situation to promote someone and do something that is inconsistent with them being promoted at the same time (Dilbert-like situations notwithstanding).

This is a made-up example, of course, but you can see that keeping data in a consistent state is not a strong point of directory services. Some directories may handle these problems better than others. You have to determine the capabilities of each directory server from its vendor.

How do you solve the problem of attributes needing to be in consistent states relative to each other, given that true ACID (see Chapter 2) is not usually supported? In our example attributes on one entry need to be consistent (executive and options), but when the problem is generalized to multiple entries it is perhaps even more insidious. You have to know who is allowed to update the data and control the updates. Data that can be changed by multiple applications, or multiple instances of the same application, need to be coordinated. Don’t assume that your directory service is a database.

Partitioning

Partitioning is the act of creating data subunits. Think of a file system tree structure and imagine that a subtree is always used by one particular department in your enterprise. It might make sense to partition this subtree and place the new partition
on a server near the department that uses the data. It might make sense to do this because you want the department (and not you) to manage that data, or because they need fast access to the data and therefore it should be close to them.

Directory service partitioning is much the same.

**Partitioning for Performance**

If there were ancient proverbs of computer science, one of them would be, “To increase the speed of your search, decrease the quantity of data you search through.” One of the advantages of creating multiple smaller partitions instead of one big partition is internal speed. (See Figure 19.11.)
Is speed important? It might be, depending on your system and application design criteria.

**Partitioning for Management**

One of the reasons to partition a file tree was to make a subtree someone else’s problem. Ah, we mean “to optimize the administration of that subtree’s data by having the owning department take care of it.” Partitioning can be a useful way to spread the management responsibility for data along organizational or other lines. (See Figure 19.12.)

**Application-Level Problems Caused by Partitioning**

Partitioning can be useful for performance or management reasons, as we have seen, but it can also cause problems, such as with referrals and search performance.

**Locating the Right Partition** The most obvious problem caused by partitioning is that the data are no longer in one place. This was, of course, exactly what we were after. As they say, “Be careful what you ask for, you just might get it.” Inevitably, the data that we partition and put somewhere else will be required by someone back in the originating location. How can they find it again? There are two possibilities.

- They just know where it went and change their applications accordingly
- They find it automatically with referrals that were inserted when the partitioning was done.

![Figure S.12  Partitions for administration](Image)
Both of these two solutions have plusses and minuses. In the first solution we are making people—instead of computers—work harder. In the second solution the directory service itself must be smart enough to create the referral, keep track of subsequent changes, and synchronize the referral accordingly. (See Figure 19.13.)

**Search Performance** Performance matters, sometimes. Infrequent searches or searches done in human time can be less performance-sensitive than searches done every few seconds and on which critical business processes depend.

Partitioning can increase performance for some users (especially if they are nearer to their data because of the partitioning) and reduce performance for other users (ones who used to be close to the data but are no longer close to it). So the performance implications of partitioning need to be weighed against the goals of the business. Knowing the usage patterns of the data can be important when making partitioning decisions.

**Referrals**

When a doctor says, “Gee, I don’t know what’s wrong with you,” you get a sinking feeling, and then you get a referral. A directory referral is the same thing. A directory server is telling you, “Gee, I don’t have your data, but you might look over there.” A referral is just an object in the directory’s store. In a way it is a meta-object. Instead of containing the data you want, it contains the address of some other server(s) that might have the data you want (it will actually contain an LDAP server URL). (See Figure 19.14.)

---

![Figure S.13 Locating a partition without referrals](image-url)
Note that referrals can be “chased” automatically by the LDAP library, or you can chase them yourself with logic in your program. The advantage to chasing your own referrals is that you can make intelligent decisions about which server or servers to use when a referral comes back with a list of several possible servers. The main advantage to letting a library chase referrals for you is that it is easier. For example, when a referral leads to another referral and back, the library will typically detect a loop. If you chase your own referrals, you will have to do that detection for yourself.

**Drawbacks of Referrals**

Referrals are a good way to “glue” one directory service to another—to make one directory name space appear within another directory’s name space, although not completely (the directory trees are not merged, for example). Here are some potential problems with referrals.

- Loops can occur in referral chains.
- Referrals can have an adverse affect on performance, especially between dissimilar directories. Each time a referral occurs, a new directory server must be contacted.
- The referred-to directory may have a different user authentication scheme, so you might need different credentials to authenticate to the other directory. It is a major pain if you have to manage multiple credentials just to access the directories to which you might get a referral—imagine keeping this credentials list up to date.

---

**Figure S.14** Locating a partition with referrals

1. Do you have my data?
2. No, but Z does.
3. May I have my data please?
Chaining

Chasing referrals is logic that is implemented either in your code or in your LDAP library. Chaining is implemented inside directory servers themselves, so when a server realizes that it needs to contact another server, it does, instead of returning a referral that you or your library can chase. Chaining is not part of the LDAP standards, so your directory server may not support it.

Putting Replication and Partitioning Together

Taken separately, replication and partitioning are powerful directory infrastructure tools. Taken together, they can be more powerful still. For example, partitioning may cause performance problems if a partition that contains some needed data is not physically nearby. So instead of partitioning and putting one partition in Poughkeepsie and another in Raleigh, we can partition the data and keep replicas of each partition, one each in Poughkeepsie and Raleigh. Now a copy is always nearby, as well as small enough to search through (or separately manage). (See Figure 19.15.)

![Figure S.15 Replication and partitioning together](image-url)
**Business Issues Drive Replication and Partitioning**

As we have seen, partitioning and replication are powerful tools for solving administration or performance problems. However, the business side to these tools should always be considered before they are used.

Again if computer science had ancient proverbs, one of them would be, “Don’t solve problems that you don’t have.” This applies equally well to business. If you consider replication or partitioning to increase performance, you should first be sure you need to increase performance, and you should be able to roughly quantify the results you wish to achieve and the impact on vital applications. Of course we don’t always know our needs in advance and that’s why we have capacity forecasts and contingency plans for growth.

**Summary**

In this chapter we discussed how replication and partitioning can help solve performance or administration problems. While both of these technical tools are useful, they are especially powerful in combination, and that’s why this one chapter discusses both.
Chapter 20

SYNCHRONIZATION AND METADIRECTORIES

Given a conflict, Murphy’s Law supercedes Newton’s.
—ANONYMOUS

In an ideal world, there would only be one directory service. In the real world, there are multiple directory services. This situation arises from distinct problems, solutions, and business processes operating within separate portions of each enterprise. A given business process implemented years ago may be still valuable. There may be no reason to replace such a process merely because of the need to streamline the encompassing enterprise operations. The goal of a directory solution is to enhance overall procedures and processes within an enterprise without ripping out its heart.

Many disparate business processes are tied to separate directory services. Chapter 22 discusses some common directory services used today. An e-mail system may use a white pages directory that encompasses the entire enterprise. An expense account workflow application may span several departments, with the workflow state kept in a directory service within the workflow application package. It is tempting to throw out extra directory services in favor of standardizing on a single service, but this is impractical for established enterprises (see Chapter 29 for a case study illustrating why this is the case).

Multiple directory services are more useful when used together, provided that there are no lasting conflicts across corresponding entities. Synchronization techniques can be used to maintain data consistency between several directory services. Update conflicts must be resolved such that the data content converges. This chapter discusses directory, synchronization and describes a new class of directory called a meta-directory, which provides the glue among other directories.

Replication

Let us review directory replication so that we may better understand synchronization. The problem of replication is straightforward in technical complexity because
the servers involved use identical schemas and service software. No schema mapping is needed—a key point.

Remember that some directory services are partitioned into trees. (See Figure 20.1.) An enterprise may structure directory services into hierarchical trees that mirror the structure of the enterprise itself. Attributes within such trees may be automatically replicated between servers of different services within the tree. (See Figure 20.2.)
Replication allows a given directory to scale well throughout an enterprise. (See Figure 20.3.) It does not address the problem of synchronizing existing disparate directory services and mapping objects between their schemas.

Synchronization

Synchronization is often confused with replication. Let us postulate a corporation whose e-mail system uses a name-and-address book that contains white pages locator information about everybody in the corporate enterprise. This corporation uses a workflow application to process time and expense accounting submissions from employees. The workflow application is bundled with its own directory service, separate from the name-and-address book directory service. Company employees submit expense account forms or forms for time worked to the time-and-expense accounting department. That department uses the workflow application to send the forms through a multiple-step accounting and approval process. The workflow application uses its directory service to reference the employee submitting the form to find the supervisor who must approve the submission.

Each of these applications uses a distinct, separate directory service that contains similar semantic user information in different schemas. Somebody must enter, modify,
and delete user information in the name-and-address book directory. Another person must maintain similar data in the workflow application. This is a wasteful duplication of effort, expensive to maintain and error-prone. (See Figure 20.4.) This scenario is simplified—real enterprises may encounter manual synchronization problems among directories where redundant administration occurs at even higher multiples.

It seems smarter to use only one directory service and one composite schema, replicated for redundancy and performance. However, this avenue presents more problems.

- One directory service must be chosen as the standard.
- A schema must be designed that is the union or mapping of all the schemas in use by the separate directory services.
- The application source code needs to be modified, but the enterprise may not own or possess all of the sources for its applications.
- Only proprietary directory interfaces may be available, to the exclusion of LDAP in some applications.
- This radical, across-the-board change must be deployed without damage to the business processes.

It seems that changing all the enterprise applications to standardize on one directory service and one schema may not be such a practical idea after all.

![Diagram showing email name and address book, corporate personnel directory, and manufacturing phone directory.](Figure T.4 Manual synchronization)

**Task:** Enter a new-hire manufacturing employee.

**Method:** Three administrators use three tools to enter information in three directories.

**Task:** Change Luanne Zyph's last name to Oakland.

**Method:** Three administrators use three tools to edit information in three directories.

**Disadvantages:** Too many people involved, error-prone, data inconsistent, too long, expensive, insecure.
Perhaps it’s smarter to leave the applications, schemas, and directory services alone, and concentrate on keeping similar or identical attributes loosely consistent across the directory services. But that’s what the humans were doing through manual synchronization. Instead we can let a computational process mirror the data from one attribute in a given service to a similar attribute in another. (See Figure 20.5.)

This technology is called directory synchronization, or just synchronization. While replication is the process of converging updates to an attribute across replicas of a given directory service containing the same schema, synchronization is about converging updates to an attribute among mappings of that attribute in other, foreign directory services across schemas. Expressed another way, replication is about updating identical-to-identical items, while synchronization involves updating foreign-to-foreign items.

For example, with synchronization, a change to a user’s office number in our hypothetical name-and-address book would eventually cause that user’s office number to be changed in the workflow directory, even though the attributes may have different names and syntaxes. This operation would be carried out regularly after some synchronization interval, by a hypothetical synchronization agent.

This is an attractive notion because no applications or directory schemas need to be modified. Business processes proceed as before, without modification. One or two manual overhead tasks provided by administrators may disappear. The reason that

![Diagram](image)

1. Clerk creates new personnel entry for Carey Grice.
2. Synchronization agent periodically awakens, notices change.
3. Agent reads enterprise entries of interest having timestamps later than its last scan.
4. Agent adds/updates entries of interest to name and address book.
5. Carey Grice’s entry is equivalent in each directory, even with different schemas.

**Figure T.5** Programmed directory synchronization
two tasks may disappear is that a user may be empowered to modify some of her own
data by providing proper access control to the attributes in question. This wouldn't
happen without synchronization because

- The access control would have to applied in each directory service
- There may not be such fine-grained access control available in each directory service
- A user may rebel at the obviously stupid notion of having to update the same data more than once

It seems that all we need is a synchronization agent. This could be commercial or a hand-crafted script or program that maps attributes between schemas, identifies the latest version of a mapped attribute, and imposes the later value on all the out-dated corresponding attribute values in the rest of the directory services involved.

**Access Control Problems**

Not all directories are flexible when it comes to access control and granularity. Users may be authorized to change their office numbers in one directory, but not in another directory because the directory implementation allows only another person to administer office numbers.

Another problem is that some specialized directories only allow “all-or-nothing” administrative rights. Domain Name System (DNS) is such a directory. There is no LDAP interface in standard DNS, but services may be advertised in DNS SRV records. Generalized information may be kept in DNS TXT records, but only a person with administrator privileges may manually change it. Such an administrator then has the right to change all of the data in DNS.

**Schema Mapping Problems**

The schemas of the participating directories need to be mapped. Attacking the schema mapping problem is key to enabling synchronization. Some confounding problems arise. One directory may define a user name attribute as “Oakland, Luanne,” while the next service uses “Luanne Oakland,” and a third may have a “surname” attribute of “Oakland” and a “firstname” attribute of “Luanne.” All of these are representations of the same “Luanne Oakland” in the different directory services, as shown in Figure 20.6.

If we look deeper, we may find similar attributes we wish to map that have different maximum lengths, different syntaxes, and different encoding schemes (for example, Unicode versus ANSI).
Replication Problems

Synchronization is a superset of replication in complexity, but not from a logical point of view. One may synchronize only the user objects in the directories within an enterprise, while entire trees of objects participate in replication. The problems multiply by the number of directory interfaces (all may not use LDAP), the number of schema mappings needed, and the number of separate access control schemes to surmount.

Another problem in carrying out directory synchronization is that there are no standards for directory replication, as shown in Figure 20.7. Several techniques

- A person, Luanne Oakland, appears in three directory schemas.
- Her name maps to differing attributes among those schemas.
- A synchronization agent must map updates across those attributes.
- The synchronization agent must have update authorization across three schemas.

![Figure T.6 Schema mapping](image1.png)

![Figure T.7 Synchronization problems](image2.png)
are currently used to carry out replication. One IETF duplication standard is called LDAP Duplication/Replication/Update Protocols (LDUP) (See http://www.ietf.org/html.charters/ldup-chapter.html.) Vendors do not store schema extensions in a standard manner. Two directory vendors may store users and passwords differently, which makes it difficult to share this information. The Directory Interoperability Forum (DIF) activity of IBM, Lotus Development Corp., IsoCor Corp., Novell Inc., and Data Connection Ltd., among others, aims to promote standards beyond LDAP Version 3 to increase directory-enabled application development.

If one directory vendor’s implementation is used in separate locations with different schemas, replication may be configured such that attributes in the two schemas are synchronized through the vendor-provided replication mechanism. The University of Michigan’s LDAP implementation currently is freely available through the OpenLDAP organization. The LDAP server is called slapd and the replication agent is named slurpd. Figure 20.8 is an example of a Linux configuration file used to synchronize attributes governed under parts of two different schemas.

![Figure T.8](image-url)
Metadirectories

The Burton Group defines metadirectory as a product that supports multiple name spaces, provides directory concatenation, supports attribute and object filtering, supports attribute and object synchronization, and provides a directory join operation and a model for information flow. (See http://www.tgb.com.) The join may be viewed as a union of attributes associated with a single object.

The glue to bind an organization together currently is not totally provided by LDAP. Chapter 28 describes an experimental Web farm that uses a DNS as a root for the site. There, DNS provides a root authority that LDAP lacks. While a hierarchy of authoritative DNS directories fastens the entire Internet together, there is no corresponding authoritative LDAP root directory. Outside referrals are manually mapped. On the other hand, Microsoft has grafted dynamic DNS to its LDAP Active Directory hierarchy, enabling trees and forests of Windows 2000 domains within an enterprise. Other LDAP developers have proposed a similar plan to piggyback LDAP atop standard DNS. This plan is currently an Internet Engineering Task Force (IETF) draft. The draft proposes that the domain component (dc) attribute would be extended to map LDAP schemas to organizations described by DNS.

Even after we locate all the directory services, there will be different implementations of directories within a large organization. At a higher level, we want to treat some subset of their content as one directory. An update of an end-user's phone number on directory should propagate to other directories organization—at least to those that store user information.

The problem is that an end-user could be identified as “Richard Russell” in a Domino Name and Address Book, show up as “Russell Richard” in a custom OpenLDAP schema, and appear as simply “Russell” in an on-line messaging application. One or more administrators would have to visit each directory and use the correct naming convention to update Richard Russell's user information. This is expensive, error-prone, and leaves the data inconsistent for too long a period. A single administrator often cannot do the job since the directories are in different parts of the directory turf. There may also be security issues involved in having one person update all the directories.

Obviously we need to be able to update an entity once and have that update propagate to the proper places in the enterprise. We need a bridge between directories that don’t have a common replication model, that have different schemas (schemata), and that have unique security requirements. A single administrator needs to be able to update Richard Russell's phone number in one place, using one convention. The correct change should be reflected in all the other participating directory services that have a corresponding attribute for Richard's phone number, mapped to whatever that attribute is called, with the syntax required, inside the proper object class container.
This bridge is called a metadirectory. The advantages of using a metadirectory are time, effort, accuracy, and consistency. A good metadirectory can actually synchronize an LDAP directory to directories that don’t have LDAP interfaces, such as GroupWise or many e-mail servers. (See Figure 20.9.)

A metadirectory isn’t a plug-and-play solution. Some careful, perhaps painful, up-front configuration is required to carry out location and attribute mappings under proper security credentials. Once the metadirectory is set up, however, the payback is undoubtedly worth the effort.

Currently there is a movement among major directory vendors to acquire metadirectory companies and to work together on interoperability standards (such as the Directory Interoperability Forum). The result should be that more directory services will know how to interoperate and synchronize with one another, not unlike the joining of mail systems and tools that occurred a few years ago.

**Summary**

Life in the directory world would be easy if there were only one directory service and one schema. But directory services are only tools for solving distinct problems not an
end in themselves. They arise at different rates, in different places in the enterprise. It is clear we must live with multiple directory services, but we need to unify the access and maintenance of these services.

Synchronization techniques seek consistency across directory implementations with different schemas. Humans have carried out this synchronization manually for years, but schema mapping along with proper access control can better synchronize attributes. Schema mapping is the key to synchronization. This may be carried out by an ad hoc hard-coded application, but this is messy and doesn’t extend to larger tasks well.

Metadirectories are a generalized approach to the problem. Once the attributes and access control across the participating directories are identified and configured, the metadirectory can appear as a single unified directory. Major directory vendors are incorporating such technologies into their products.
his chapter discusses the differences between a directory service API and a directory service protocol, as well as the origins of directory protocols and APIs. Various protocols are briefly discussed, and a number of directory programming interfaces are introduced.

Differences between APIs and Protocols

The protocol used to access a directory and the programming interface used to access a directory are often lumped together. The distinction between these two interfaces is subtle and, until recently, has not made much difference because the two interfaces were usually closely tied together.

The emergence of LDAP as an industry-standard protocol, however, has clarified the distinction between protocol and programming interface. Protocols isolate directory implementations from users and programming interfaces isolate programs from protocols. Many programming interfaces map closely the protocols that they isolate. To understand programming interface, then, we must understand the underlying protocols.

A protocol is a definition of data formats that can flow over a communications mechanism. Data communications protocols are usually described in a language that can be resolved down to byte-by-byte data flows over a data communications mechanism. The protocol is defined in terms of requests and responses with very clear formats for the data that appears on the communications wire.

A programming interface, on the other hand, defines a set of subroutines, classes, methods, or interfaces in one or a number of programming languages. The programming interface provides a simpler way to use a protocol than writing low-level communications programs to read and write data from a data communications
mechanism. Depending on the target programming language, the programming interface may be defined in terms of structures (data types) and subroutines, as in the LDAP C language API, or in terms of a set of abstract interfaces, as in the Java Naming and Directory Interface (JNDI).

In some cases, the programming interface and protocol are closely related. Two examples of this are DCE CDS, where the programming interface is defined in terms of a remote procedure call (RPC) interface definition language (IDL), and OMG COSNaming, where the programming interface is defined in terms of a distributed object interface in OMG IDL.

**Protocol**

As briefly noted, a protocol provides a means of separating a requester of information from the holder of this information. For a directory service, the provider and consumer of directory content are physically separated, allowing each to leverage its strengths independently. This means the directory can be accessed from systems other than the system on which the service is held, making the information available to many more systems in a network. Figure 21.1 shows how a protocol separates a directory server from a directory application.

Until recently, different directory service implementations came with different access protocols. One reason LDAP is so important is that the industry has agreed on it as a
common protocol for retrieving and modifying information in a directory service. The strength of LDAP is that directory server implementations can be built independently of directory client exploiters and vice versa. Not only does this allow different directory server providers to create unique value-add features, it also allows applications to exploit directory services without requiring one or another specific directory server implementations. LDAP facilitates directory service exploitation by eliminating the requirement for directory server-specific programming in directory-exploiting applications. Removing the dependence of an application on a particular directory server implementation is critical for pervasive directory exploitation and LDAP has removed this impediment.

**Programming Interface**

While a protocol allows directory applications to be written independently from the directory server implementation, writing low-level communications code to encode and decode wire protocol data units is a tedious, error-prone, and time-consuming task. In order to isolate directory application programs from the intricacies of communications protocol, protocols are usually accessed using a programming interface. The programming interface takes care of the details of setting up, managing, using, and tearing down the communications path with the directory server so the directory application can concentrate on using the directory service, not the directory access protocol. In summary, a programming interface provides a simpler way to access information than using the communications protocol itself. Figure 21.2 shows how a programming interface isolates the directory application from the directory access protocol.

Which programming interface you choose depends on the application and how the application will use the directory service. Some programming interfaces act as thin veneers on top of the low-level communication protocols that they isolate. The LDAP C language programming interface is a good example of this type of programming interface. The core set of subroutines in this programming interface have parameter lists that map very closely to the communications protocol data units of LDAP.

While the simple LDAP C language interface can be used to provide powerful applications, the interface is too low-level for some applications. For other applications, the set of operations to be performed using a directory does not easily map to the LDAP communications protocol. An example of this is the use of a directory for publishing and finding distributed object instances in a distributed object operating environment. This use requires a general mechanism for associating a name with a
distributed object instance. When a distributed object instance is to be invoked, the location of the distributed object is looked up in the name space of distributed objects. Once the distributed object instance is located, a proxy object is created to represent and communicate with the distributed object instance. The result of a lookup operation, then, is a handle to the proxy object that represents the distributed object instance. This type of programming interface is defined in the OMG IDL definition for COSNaming services.

Even with the emergence of LDAP as an industry standard, other protocols continue to be used in many applications. To address the complexity of writing applications to use these different protocols, new “agnostic” programming interfaces have been created, frameworks under which specific directory access protocol providers can be built.

The application program is isolated from the directory access protocol used by the programming interface framework. This allows directory applications to use different directory servers as well as different directory access protocols, depending on how the applications are configured and deployed. This is a powerful means of isolating directory applications from the directory service implementations that they exploit. Examples of directory access frameworks are Active Directory Services Interface (ADSI) and Java Naming and Directory Interface (JNDI). Service providers have been written to access DNS, the local file system, Novell NDS, OMG COSNaming, and LDAP servers, using directory-specific protocols. Figure 21.3 shows how, with an API framework, an application can use one interface to access multiple protocols.
Origins of Directory Protocols and Programming Interfaces

Directory services have existed in various forms for a long time. However, until the industry acceptance of LDAP, the directory service, access protocol, and programming interface were always tied together as a package. In order for a particular application to exploit a directory service, the application necessarily had to depend on a particular directory service implementation. This inhibited application exploitation of a directory service to enhance manageability and eliminate duplication. Indeed, many applications instead created yet another directory of information, specific to the application.

Novell Directory Service (NDS) and Network Core Protocols (NCPs)

When Novell introduced the Novell Directory Service (NDS) to the Netware product to replace the Netware Bindery Services, the X.500 set of standards was just emerging. NDS is based on the X.500 set of standards for the data model that the directory implements. NDS is viewed by many to be ahead of its time.
Along with the definition of the directory service, Novell created a protocol for accessing the directory over IPX networks by extending the Netware Core Protocols (NCPs) used for communication between Novell clients and Netware servers. The protocol for accessing NDS was NCPs over IPX.

In addition to defining NCPs for accessing NDS information, Novell extended the client programming interfaces to provide application programs with an interface for accessing NDS information. These APIs, all of which begin with the NWDS prefix, are aligned with the set of operations that can be performed on information in NDS. Programming interfaces exist for accessing, modifying, adding, and removing information as well as accessing, modifying, adding, and removing schema information that is held in NDS. In addition, programming interfaces exist for logging into NDS and for authenticating to NDS.

Novell has worked diligently to support LDAP as an access protocol over TCP/IP for accessing NDS and hence support programming interfaces that use the LDAP protocol to access NDS.

### Distributed Computing Environment (DCE) and Cell Directory Services (CDS)

When the Distributed Computing Environment (DCE) was introduced by the Open Software Foundation (OSF), it included a remote procedure call (RPC) environment as well as a directory service, security service, and time service for supporting the RPC programming model. This programming environment represented one of the first open programming environments to cover directory and security aspects across a network of connected systems. This was done by a consortium of companies—one of the first such implementations ever accepted by multiple companies. For all its merits, however, DCE was very much an all-or-nothing proposition. DCE used its own services (RPC) to provide the directory, security, and time services that RPC services themselves depended on. In order to exploit many of the services of DCE, you needed to use the RPC programming model. This integration, while providing a very function-rich environment, has hindered DCE's acceptance across the computing industry.

The protocol used to access the directory in DCE, called Cell Directory Services (CDS), is defined by the encoding of the RPC parameter lists sent over a TCP/IP communications path. The DCE IDL definition of the RPCs defines the protocol used to access CDS.

There are two programming interfaces to the information in CDS. For creating, modifying, and removing DCE RPC interface definitions, called RPC bindings, there are C language programming interfaces in DCE that are part of the DCE RPC programming model. These interfaces all have a prefix of rpc_ns. The second pro-
gramming interface to CDS is the XDS/XOM programming interface. In DCE, XDS/XOM can be used to access both CDS and the Global Directory Service (GDS), if GDS is part of the DCE installation.

## X.500 and Directory Access Protocol (DAP)

The X.500 set of standards is the basis for many of the data models used in directory service implementations today. While very complex, the X.500 set of standards presents a directory data model that is hierarchical, extendable, and rich in the set of data types it supports. The data model is generic and extendable such that the model can be applied to many applications’ data storage requirements.

As described in Chapter 4, the X.500 set of standards contains a detailed description of the set of protocols used between X.500 clients, called directory user agents (DUAs), and X.500 servers, called directory service agents (DSAs). The protocol used between DUAs and DSAs is called the directory access protocol (DAP). This protocol was originally defined to be carried over the OSI protocol stack but has since been defined to be carried over TCP/IP. LDAP was first created as a simplified version of DAP.

Many programming interfaces have been defined for accessing X.500 directories. These programming interfaces are provided in the form of a DUA implementation, usually provided by a X.500 DSA vendor.

At one point an attempt was made to define a standard programming interface to an X.500 directory. XDS/XOM was a programming interface defined by the X series of recommendations; the programming language was not defined but the set of parameters for these interfaces was defined. DCE contains an implementation of XDS/XOM for the C language. The XDS/XOM C interface makes use of a set of complex structures to represent information passed to and retrieved from a directory service.

## Popular Directory Protocols

A number of directory access protocols are in use today. Many applications use these protocols without their users even knowing about it. With the industry acceptance and exploitation of TCP/IP networking, most of these protocols flow over TCP/IP communications paths.

---

1. XDS/XOM is described in the section on DAP and XDS/XOM.
LDAP

LDAP has won industry acceptance as the access protocol to use for directory servers that implement a data model based on the X.500 data model. Because the X.500 data model can be applied in many ways, many existing directory services can make their information appear in a X.500-defined format. LDAP represents the de facto standard protocol that can be used to access and update information in many directory services.

Domain Name Service (DNS)

The protocol used to access information in a domain name service (DNS) was originally defined in IETF RFC 883 and has been enhanced over time. At the time of this writing, IETF RFC 2535 represents the latest set of enhancements to DNS and the protocol used to access and update DNS. (See Chapter 4 for a discussion of DNS.)

The programming interface to DNS is defined by a DNS client called a resolver. Typical programming interfaces to a resolver allow IP address lookups based on DNS host name or other lookups based on DNS host name. In the C language, for example, the `gethostbyname()` function is considered a programming interface to DNS.

Novell Directory Access Protocol

Novell directory access protocol (NDAP), really the NDS NCPs flowing over either TCP/IP or IPX, represents a second popular directory access protocol. Most Netware users rarely see NDAP in their environment because it is hidden behind the Novell client and Novell administration programs that use it. Novell Netware environments are used today in many computing environments.

COSNaming over IIOP

The directory protocols described to this point are based on mainly a procedural approach toward accessing a directory service. The directory service is expected to respond to queries for information and return the results of these queries. In contrast to these protocols, a complete protocol for distributed object communications is defined by the Internet inter-ORB protocol (IIOP). IIOP defines a protocol for distributed objects to communicate with one another much as remote procedure call protocols do. An important part of distributed object environments is the capability for locating and instantiating local proxy object instances to communicate with remotely hosted distributed object instances.
The Object Management Group (OMG) COSNaming interface defines a set of distributed object methods for associating distributed object instances with names in a name space, looking these instances up in the name space, and instantiating proxy objects in order to invoke methods on the distributed object instances. Since this interface is defined by the OMG IDL language, the interface also constitutes a protocol. This protocol represents an additional directory access protocol tuned to the requirements for distributed object environments.

**Popular Directory Programming Interfaces**

The rising popularity of the LDAP protocol as a separator between directory server implementations and directory applications has created an environment where directory programming interfaces can be developed independently from any particular directory server implementation. Since there are many popular programming languages—C, C++, Perl, Java, and Tcl/Tk to name a few—and many applications written in these languages can benefit from directory exploitation, a number of programming interfaces have been developed that use the LDAP protocol.

Common themes run through all of these programming interfaces, but each has its own set of features. In general, each of the programming interfaces attempts to exploit the natural characteristics of the programming language from which it is callable. For example, in the programming interfaces defined in Java, `java.lang.String` objects are used for input and output parameters, where null-terminated character strings are used in the C language interfaces.

In all cases, there are three areas where the programming interfaces are challenged. The input to the add operation in all of the programming interfaces is a complex set of information. This is to be expected, since the underlying directory model (that is, a directory entry has a name and a set of one or more attributes, each of which as a set of one or more values) is complex. The input to the modify operation is also a complex set of information since multiple attributes can be modified (added to, deleted from, or replaced) within a single modify operation to the directory. Third, the results from a query operation (read, list, or search) is a complex set of information. The results of a search operation can contain information from multiple entries, each having multiple attributes, which, in turn have multiple values.

**LDAP C**

The LDAP C language programming interface was the first programming interface defined to simplify the use of the LDAP protocol. The LDAP C language interface is a set of C functions that allow a program to open a connection to an LDAP server.
Once a connection is open, the application can bind (authenticate) to the directory server. C functions are provided to add, modify, delete, and rename entries in the directory. A set of search functions allows subtree, one-level (list), and base (read) search operations. The C language API allows the application to perform asynchronous operations. In asynchronous mode, a function is invoked to send the request over the protocol. After sending the request, the function returns to the caller. A function call is then made by the application in order to wait for a response from the directory server, sent to indicate the result of the previously sent request.

An example of invoking the search operation using the LDAP C language API is shown in Code Example 21.1. The search performed is a subtree search for all `inetorgperson` objects that contain a `mail` attribute and the only attributes returned are `mail` and `cn` (commonName).

The LDAP C language programming interface is relatively simple yet quite powerful with its support for asynchronous operations. Refer to Appendix A for a mini-reference on using the LDAP C language programming interface.

**JDAP**

JDAP is a programming interface that attempts to provide to Java programmers what the LDAP C language interface provides to C and C++ programmers. JDAP encapsulates the connection to an LDAP server in an `LDAPConnection` Java interface. This interface defines methods for all the basic LDAP operations, including add, modify, search, and delete. Java string objects are used as input parameters to the methods and add, modify, and search results information is provided in Java object instances, helping to simplify the handling and parsing of these complex input and output formats.

An example of invoking the search operation using the JDAP API is shown in Code Example 21.2. The same search is used as was defined for Code Example 21.1.

```java
attrs[0] = "mail";
attrs[1] = "cn";
attrs[2] = NULL;
rc = ldap_search_s( ld, "ou=dept1, o=widgets", LDAP_SCOPE_SUBTREE,
"(&(objectclass=inetorgperson)(mail=*))", attrs, 0, &res );
```

**Code Example 21.1** Invocation of a search operation using the LDAP C language interface
The JDAP interface provides an easy to use interface in the Java language for accessing directory information using the LDAP protocol.

**PerLDAP**

PerLDAP is a programming interface that provides to Perl programmers what JDAP and LDAP C provide to Java and C/C++ programmers, respectively. A Perl package is defined with methods that are related to the LDAP protocol elements. Methods are provided for performing all operations defined by the LDAP protocol. Because of Perl's rich string and list-processing capabilities, specifying input parameters to some of the PerLDAP methods is easier than with the LDAP C language interface.

An example of invoking the search operation using the PerLDAP API is shown in Code Example 21.3. The same search is used as was defined for Code Example 21.1.

The PerLDAP package provides easy access to LDAP for Perl programs.

```java
String attrs[] = new String[2];
attrs[0] = "mail";
attrs[1] = "cn";
LDAPSearchTree results = conn.search( "ou=dept1, o=widgets",
LDAPv2.SCOPE_SUB,
"(&(objectclass=inetorgperson)(mail=*))", attrs,
false, NULL);
```

**Novell Development Kit (NDK)**

The Novell Development Kit (NDK) contains functions for accessing Novell NDS using Novell NCP protocols. While this is not an open interface, these programming interfaces are used in many environments today. Programming interfaces exist
for creating a connection to NDS, logging into and authenticating to NDS, accessing and modifying information in NDS, and terminating the connection to NDS. All NDS programming interfaces begin with the prefix NWDS.

An example of invoking a search operation to NDS using the NDK NWDSSearch API is shown in Code Example 21.4. The same search is used as was defined for Code Example 21.1. Note that the naming model used is slightly different due to the string formats for names in NDS.

The NDS access functions use the NDAP protocol to communicate with the NDS server. Novell is moving toward the use of open protocols, including LDAP, to access information in NDS directories.

```
Buf_T *filter;
Buf_T *attrs;
Buf_T *results;
Filter_Cursor_T *filtCursor;
NWDAAllocBuf( DEFAULT_MESSAGE_LEN, &filter );
NWDAAllocBuf( DEFAULT_MESSAGE_LEN, &attrs );
NWDAAllocBuf( DEFAULT_MESSAGE_LEN, &results );
NWDAAllocFilter(&filtCursor);
NWDAAddFilterToken(filtCursor, FTOK_BASECLS, NULL,0);
NWDAAddFilterToken(filtCursor, FTOK_AVAL, "inetorgperson", SYN_CLASS_NAME);
NWDAAddFilterToken(filtCursor, FTOK_AND, NULL,0);
NWDAAddFilterToken(filtCursor, FTOK_ANAME, "mail", SYN_CI_STRING);
NWDAAddFilterToken(filtCursor, FTOK_EQ, NULL,0);
NWDAAddFilterToken(filtCursor, FTOK_AVAL, ",", SYN_CI_STRING);
NWDAAddFilterToken(filtCursor, FTOK_END,NULL,0);
NWDAPutFilter( ctx, filter, filtCursor, NULL );
NWDAPutAttrName( ctx, attrs, "mail" );
NWDAPutAttrName( ctx, attrs, "cn" );
rc = NWDSSearch( ctx, "ou=dept1.o=widgets", DS_SEARCH_SUBTREE,
                TRUE, /* de-reference aliases */
                filter,
                DS_ATTRIBUTE_VALUES,
                FALSE, /* not all attributes */
                attrs,
                &iterHdl, 0, &numSearched,
                results );
```

**Code Example U.4** Invocation of a search operation using the NWDSSearch interface to access NDS
COSNaming

The interface defined by OMG COSNaming is very different from the programming interfaces described in the previous sections. Instead of mapping to protocol elements underlying the communications between the programming interface and the directory server, the COSNaming interface defines a set of methods for associating (binding) a name to a distributed object instance, disassociating a name from a distributed object instance, and looking up distributed object instances and instantiating proxy object instances that represent the distributed object instances that were found during the lookup operation.

The base set of methods defined by COSNaming are `bind()` (associate a distributed object instance with a name), `unbind()` (undo what was done by the `bind()` method), and `resolve()`. These interfaces allow distributed object instances to be published, found, and later removed from a distributed object name space. All names used in this interface are relative names, based on the instance of the naming context that is used to invoke the methods defined in the interface. An obvious question is how the first (or initial) naming context instance is created by the application. Creation of the initial context is performed by invoking a method defined by the object request broker (ORB) in which the distributed object application is running.

An example of invoking the `resolve()` method in the OMG COSNaming interface is shown in Code Example 21.5. In the example, the current naming context is used to look up a context with relative name “widgets/dept1.”

```
obj = ctx->resolve( "widgets/dept1" );
```

<table>
<thead>
<tr>
<th>Code Example 21.5</th>
<th>Invocation of a lookup operation using the OMG COSNaming interface</th>
</tr>
</thead>
</table>

The OMG COSNaming interface is important both for distributed object environments as well as for its use as the basis for the interfaces defined by the JNDI programming interface.

JNDI

JNDI represents an attempt to merge the characteristics of a distributed object lookup service with a generalized directory access interface. JNDI accomplishes this by defining two interfaces: `Context` and `DirContext`. Methods defined by the `Context` interface map to the methods defined by the OMG COSNaming interface. Methods defined by the `DirContext` interface map to the set of operations that can be made to a directory service: add, modify, delete, rename, and search, to name a few.
Defining Context and DirContext as Java interfaces allows different implementations of these interfaces to be created. The JNDI framework is an environment where multiple service providers can provide instances of objects that implement the Context or DirContext interfaces. As the DirContext interface extends the Context interface, a JNDI service cannot provide DirContext without providing some implementation for the methods defined by the Context interface. With JNDI, then, there is a provision that every directory entry could potentially contain information that associates a distributed object instance to the directory entry. This natural merge of distributed object naming and generalized directory services is a powerful concept.

The format of names provided to the Context and DirContext methods is dependent upon the particular service provider that has provided the object instance that implements the interfaces. The JNDI framework defines mechanisms for transitioning between naming systems, that is, service providers, by means of junctioning two name spaces together. In practice, however, applications typically use a single service provider and naming model.

Code Example 21.6 shows an example of invoking the same lookup operation as described in Code Example 21.5 using the JNDI Context lookup method.

```
String attrs[] = new String[2];
attrs[0] = "mail";
attrs[1] = "cn";
SearchControls ctls = new SearchControls();
ctls.setSearchScope(SearchControls.SUBTREE_SCOPE);
ctls.setReturningAttributes( attrs );
NamingEnumeration namesEnum =
    ctx.search( "ou=dept1, o=widgets",
                "(&(objectclass=inetorgperson)(mail=*))",
                ctls );
```

**Code Example U.7** Invocation of a search operation using the JNDI DirContext interface

Code Example 21.7 provides an example of invoking the same search operation as was described for Code Example 21.1, using the JNDI DirContext search method.
JNDI provides a powerful, multipurpose programming interface for Java programmers accessing distributed objects as well as information in a directory service. With Java’s popularity, JNDI is one of the most popular interfaces for accessing directory information. See Appendix B for a mini-reference on using the JNDI DirContext interface to access a directory service.

Active Directory Service Interface (ADSI)

Active Directory Service Interface (ADSI) provides a set of programming interfaces for accessing directory information from Microsoft-based operating systems and programming environments. Like JNDI, ADSI provides a framework under which different directory service providers can be accessed. ADSI provides interfaces to add, modify, search, and delete information in a directory service.

Code Example 21.8 provides an example of invoking the same search operation as was described for Code Example 21.1, using the ADSI search method.

ADSI suffers from being a set of interfaces specific to the Microsoft programming environments.

Summary

This chapter has described the differences between a directory access protocol and a directory application programming interface and why the definition of both is important. Popular directory access protocols and programming interfaces were also introduced with descriptions of some of the features of these interfaces.
There are several directory access protocols commonly used today, including BIND for DNS, NCPs for Novell NDS, DAP for X.500, and RPC for DCE CDS. The de facto standard directory access protocol is LDAP.

There are several programming interfaces available for using the LDAP protocol to access information in a directory service. These programming interfaces exist for C, C++, Java, and Perl environments.
Chapter 22

DIRECTORY IMPLEMENTATIONS

I coulda had class! I coulda been a contenda!
—ON THE WATERFRONT

This chapter briefly discusses a few available directory services. The implementation may support one or more protocols and APIs, in addition to features necessary to support directory services that are over and above the APIs and protocols. Not all directories are LDAP-enabled, as we shall see.

Commercial Implementations

IBM SecureWay Directory

The IBM SecureWay Directory is available on several platforms.

- IBM Netfinity Server
- IBM OS/400
- IBM RS/6000 AIX
- IBM OS/390
- Microsoft Windows NT
- Sun Solaris

Other features are SSL, client-side certificates, administration GUI for schema management, and plug-in support for user exits. The directory is bundled with solutions or applications. A nonexpiring, license-limited function is included on the CD.

Novell Directory Services (NDS)

Novell Directory Services (NDS) is one of the oldest directory implementations extant. It uses objects to represent Novell network resources and maintains them in a hierarchical directory tree. The LDAP protocol was available as an add-on through
LDAP Services for NDS in early 1997. This NetWare Loadable Module (NLM) enables authorized users to access or modify information through LDAP. An NLM is a DLL or shared library for the Novell network operating system. The directory may be securely accessed or updated from a Web browser over an intranet or Internet.

NDS for NT version 2.01 enables Windows NT domain users to be combined with NDS users and then migrated to NDS. Replicas may be stored on NetWare servers or an NT domain controller. Objects may be added, modified, or deleted in batch mode using LDIF files. A utility called ConsoleOne may be used to interactively manage objects. The Schéma utility is used to manage schema and to set attribute entry policies. Synchronization with other directories is enabled through a third-party product called NetVision Synchronicity.

Novell has a Java API called SSL for Java. It enables the development of applications that provide security over the Internet and intranets. SSL for Java provides a security layer between TCP/IP and other application protocols such as HTTP and FTP.

Authentication and encryption in the client and server extend SSL technology to Java 1.1 and Java 2.

**Netscape Directory Server**

The Netscape Directory Server was written with LDAP as its core protocol. This server is available as a trial download. It includes an LDAP client, SDKs, HTTP session management, and CGI APIs. A command-line client shell is also included. The directory SDK includes common schema objects for users and groups, and other common objects.

A Netscape metadirectory supports two-way synchronization with many database, e-mail and network operating system (NOS) products. It supports a join operation to include sources from different directories into a single result.

**Microsoft Active Directory**

Microsoft Active Directory is probably the most important feature of Microsoft’s new operating system, Windows 2000. Active Directory awareness is part of the core operating system. The directory is the enabler for many Windows 2000 features, such as Intellimirror data that roams with users, and policy-based management. A server may optionally be promoted to a domain controller. This action

installs Active Directory and inserts objects representing the domain, users, and other important entities.

**Lotus Domino Name and Address Book**

The Domino Name and Address Book is an LDAP directory offering from IBM, the parent company of Lotus Development Corporation. It may be used as the foundation of a general-purpose directory infrastructure in the enterprise.

This directory, the heart of Lotus Domino, maintains centralized, intermittently offline management of users, connections, roles, and groups, among other entries, for the Notes infrastructure. State-of-the-art security and replication technologies enable customized mixes of centralized or decentralized administration. Multimaster replication is an important characteristic.

A directory catalog is a compressed concatenation of all or some of the directories in the entire enterprise. It is akin to the SysVol in Microsoft's Active Directory. The Directory Assistant, also known as the Master Address Book, dispatches queries to the appropriate domain based on hierarchy if the master replica is unavailable. It is responsible for referrals to foreign LDAP-based directories such as NDS.

The Domino 5.0 directory administration and replication will integrate with the Windows 2000 Active Directory. Synchronization of user and group updates with the Active Directory will be supported. A unified login between Windows 2000 and Domino will be available.

**Banyan Vines**

Banyan Vines is a network operating system (NOS). Within Vines is a directory named StreetTalk used to integrate services, applications, and users within the system. A single-system image and a single point of administration for geographically dispersed organization is featured.

**An Open Source Implementation**

The original University of Michigan LDAP directory implementation is called slapd. The replication service is called slurpd. Finally, the X.500 gateway is known as LDAPD. The OpenLDAP Foundation (see http://www.openldap.org) distributes it.

---

2. There is an expanded technical article on the Domino directory included on the CD under this chapter.
as open source. This full-featured directory source is freely downloadable along with semi-automated build and installation procedures. A test suite gives us confidence that a build works. A build may be carried out for several platforms at once. Unix man pages may be installed on Linux systems.

Chapter 28 demonstrates the OpenLDAP products for use in a prototype Linux/Apache Web farm. The application finds its backend DBMS dynamically through the directory.

**Specialized Implementations**

**Domain Name Service (DNS)**

The basic function of the name server is to provide information about network objects by answering queries. This directory service does not use LDAP, but it may operate in concert with LDAP services. For instance, Microsoft Active Directory uses DNS at the top level of granularity in its directory implementation.

The specifications for the DNS name server are defined in RFC1034, RFC1035, and RFC974. RFC 819 covers the original structural ideas of DNS.

The DNS has three major components:

1. The domain name space and resource records in a tree-structured name space.
2. Name servers hold information about the tree structure. Each has authority over a part of the domain tree. The name server may forward parts requests to other name servers to form a distributed database.
3. Resolvers are like stubs that extract information from name servers on behalf of client requests.

DNS has many types of data records aside from name-to-IP-address mappings: TXT records contain any type of text string. SRV records advertise a service, as in an LDAP yellow pages directory. An interesting feature of later DNS servers is that they can be set to rotate a name mapping through several addresses round-robin style. We use this feature in our proof-of-concept Web farm in Chapter 28.

---

3. RFCs may be found from RFC index sites. Try one of these: [http://www.rfc-editor.org/rfcssearch.html](http://www.rfc-editor.org/rfcssearch.html), [http://andrew2.andrew.cmu.edu/rfc-front.html](http://andrew2.andrew.cmu.edu/rfc-front.html), or [http://www.cis.ohio-state.edu/hypertext/information/rfc.html](http://www.cis.ohio-state.edu/hypertext/information/rfc.html)
Service Location Protocol (SLP)

Service Location Protocol (SLP) is an Internet Engineering Task Force (IETF) standards-track protocol described in RFC 2165. As its name implies, SLP is used to aid in the discovery and utilization of distributed services. It is not a distributed name resolution service like DNS. A server may use it to dynamically distribute workload to the server having the smallest workload. The SLP specification defines a methodology to carry out these service functions.

- Service registration
- Service advertisement
- Service query

A form of agent technology is part of SLP. A user agent (UA) requests service information on behalf of applications. A service agent (SA) carries out service registration and service advertisement. The directory agent (DA) collects data from service agents. User agents on an intranet later request this information by specifying its attributes and the result is returned to the application in a URL. The IBM eNetwork Communications Server for Windows NT Version 6.0 supports SLP load balancing.

Summary

We have discussed some commercial directory implementations, and one open source full-featured LDAP implementation that acts as a reference LDAP release for some of them. There are more commercial implementations than those mentioned here. Synchronization and metadirectories, covered in Chapter 20, may be needed to provide a unified directory image to users and applications in environments where several directory services exist.
Directories are by nature distributed, so the case studies in this section focus on distributed and networked computing. The first three parts of this book presented enterprise software background, technologies, and key features of directories. This section puts all of that in a mental toolkit and uses it to solve problems. Each chapter in this section starts with an enterprise problem and crafts a directory-based solution.

Each case study also includes the shortcomings of the solutions developed. This is important because every engineering solution has unique design points. The acceptability of a solution is not whether it is the ultimate piece of code but whether it meets its design criteria and the needs of its users. Understanding the limitations of any design, whether in software or in any other kind of engineering, is extremely important and enlightening (and at least a little bit fun).

The case studies in this section are not complete enterprise solutions. Each case study illustrates a specific problem and the beginnings of a solution. Each example presented here was written specifically to point out the direction for solving an enterprise problem, but the programs presented are not enterprise applications running at your local bank.
his case study uses many of the techniques and concepts from earlier chapters to show how to support roaming users with an LDAP directory service. Roaming is commonly associated with road warriors, job sharing, or other professionals who use computers to do their jobs, but who may not always use the same computers. A good example is an open office, in which any salesperson can enter and sit down at a computer. They won’t always sit down to the same computer every day; in fact they might not even visit the same office building every day. (See Figure 23.1.)

Roaming is not for everyone. In particular, software engineers and IT managers, the people most likely to read this book, are unlikely to roam. Think about your company, though: bank tellers or data entry people may work from any terminal head (whether full PC or some other computing device); office managers may job-share; salespeople may move around. Roaming is a commonly desired feature in many job situations. The applications used by roaming users should make it easy to roam.

Even if your job doesn’t lend itself to roaming, how often have you gotten a new machine and dreaded spending the days necessary to set it up the way you like it? Ever wonder why the applications and state of your old laptop can’t just be downloaded into your new laptop? Maybe if you put the new machine under the old one and leave them that way overnight, the bits will just migrate downward (toward neutral earth). Well if those bits were in a directory, they could migrate downward into your new machine and you wouldn’t have to spend days setting it up. The “new machine” case is a superset of the roaming user case that we are about to examine.

How Directories Enable Roaming

Directories can store the personalized application settings used by a roaming employee. Wherever the employee roams, he has his settings. The application itself may follow the user (for example, wherever you sit down and log in, your version of Hunt the Wumpus is there), or only the personal preferences associated with the application may appear (for example, HTW’s sound on/off setting) and
other data (for example, your current game state and score), or any combination of these. In this case study, we show how application settings, customer data, and other sales related information can follow a user throughout a fictional decentralized enterprise utilizing a directory service.

Because this is the first directory case study, we won’t try to solve every possible problem. Instead, we will try to make roaming possible with a clear, short, but still useful example.

**Description of the Roaming Problem**

The Little Fuzzy Ferret Friends Company (L3F—a trademark safe name if there ever was one) makes and sells gourmet ferret food. The salespeople are typically on the road, talking to distributors and large customers about Weasel Wieners, and have no single office. Instead, L3F has roaming enabled sales offices in many cities. Any salesperson

---

**Figure W.1** Road warriors or job-sharers share computers, but don’t want to share each other’s personal settings and data

---
can visit any sales branch office while traveling, but obviously they tend to visit only the branches nearest their territory. Each sales office is minimally staffed to support these roaming road warriors and other local sales and marketing support functions.

Corporate headquarters is located in Wabash, Indiana. At HQ, the L3F company has its main offices, main IT department, and like all sales offices it is equipped to support roaming salespersons (HQ is also the sales office for the Wabash region). (See Figure 23.2.)

All of the L3F sales offices and its corporate headquarters are connected with an intranet. The intranet connections are reasonably reliable and fast. We won’t be pushing huge amounts of data around for this application, although there are probably other corporate applications as well (and there is future growth to consider, since the ferret food business is booming).

Figure W.2 L3F runs its ferret food business from Wabash, the hub of the gourmet pet food industry in northern Indiana.
The Sales Application (TSA)

The problem we want to solve concerns supporting the L3F roaming sales force. We want to make their jobs easier by building an application that

- supports the roaming salesperson’s job and enables him or her to sit down at any computer connected to L3F and have that computer appear to be “theirs.” The state of the application will follow the user as well as other data needed to make sales.
- is always, or nearly always available, so that sales are not lost unnecessarily.
- does not require the salespersons to be computer science graduates, as well as experts in ferret food and sales.

Since everything needs a name, the fictional application will be called The Sales Application or TSA. TSA will allow the sales force to do their jobs more effectively. TSA will also help us maintain the following data. (See Figure 23.3.)

- The salesperson’s personal settings for TSA itself. We will invent a few trivial settings, such as window position, just for examples. This data is specific to each user.
- A product list (a catalog) and descriptions. These are the products that the salespersons are trying to sell. This data is not specific to any one user.

![Diagram](image.png)

**Figure W.3** Various pieces of data that TSA maintains
Customer and distributor contact lists. There are two different lists. One contains the contact information that each salesperson personally uses (things like, “Played golf with Ralph of MegaPet4U on Thursday 7/8 and he remarked that Ferret Fois Gras tastes pretty good. I didn’t ask how he knows this.”). The second list is of all customers and which salesperson is currently responsible for each customer.

Schema

There are a few obvious things that TSA needs to access in the directory.

- A salesperson object
- A contact list associated with each salesperson object (the salesperson’s contact data and notes)
- A customer object
- A customer list not attached to any salesperson (the list of which salesperson “owns” which customer)
- A TSA object (the application itself)
- A product object
- A product catalog

The salesperson, customer, and product objects clearly need to be represented as directory objects (with a schema class), but what about the customer list, contact list, product catalog, and TSA object? The customer list and product catalog are really just collections of customers and products respectively, so they can be constructed from existing objects instead of making new objects to represent them. The contact list could be an object associated with a particular salesperson, or it could be attached to the salesperson directory class (as an auxiliary class, or just new attributes added to the salesperson class). We’ll choose to create a contact object to make them easier to create in and delete from TSA. The TSA object is really just an application object, which many directories will already have.

TSA Schema Class Objects

After scrubbing the initial list, here is the new list of objects that we need. Either these objects are already in our directory service, or we will need to modify our directory schema to include them. (See Table 23.1.)

User object—ideally, we would modify the existing user objectclass by adding an auxiliary class called BFSalesPerson. Recall that an auxiliary class can be mixed in to other classes to add attributes. BFSalesPerson would tailor the existing
user class with attributes that are specific to being a salesperson for L3F. The case study code creates a new structural class instead of an auxiliary class, just to make it easier (schema changes can’t be reversed in the directory server used in this example). Notice that in our naming convention the first letter is lowercase; this is just a convention and you should consult your own directory service for your naming conventions.

Contact object—We will assume there is no such object already and will create a new schema class for it, called L3FCustomerContact. These L3FCustomerContact objects will be aggregated into lists shown to salespersons using the TSA GUI.

Customer object—We will create a new schema class called L3FCustomer to represent our customers.

Application object—There is probably already an application object in our directory service. However, we don’t really want an application object that represents the TSA application. We want an application object that represents a specific instance of TSA for a particular user. We really just want a place to store TSA data that is specific to a user (the customization data, like user preferences). So we will just add these customization attributes to the L3FSalesPerson class instead of creating a new class.

Product object—There might be a relevant object class in your directory to represent these, but for good measure we’ll create a new one called L3FProduct. The product catalog is just the collection of all L3FProduct objects and will be constructed as needed. The catalog can be presented in different views by TSA (alphabetical, by specialized ferret topic, etc.).

**TSA Schema Attributes**

Now we should enumerate the attributes we need for each class and describe how they will be used. We will create new attributes for each of the new schema classes. In
some cases, attributes that could be used will already exist; we should use existing attributes wherever possible. For this example, we have a small number and we will just define them all to keep it simple.

Each of the new attributes will be listed in a table that corresponds to the new schema class. Some important information is not listed here but can be found in the code files included on the CD (for example, OIDs, syntaxes, and whether each attribute is single- or multivalued, required or optional). See Tables 23.2 through 23.5.

The Directory Information Table

Now that we have objects to represent all of the directory entries that TSA needs, we have to decide where these objects can be stored in the directory information tree (DIT). Every directory schema class has an attribute that defines where instances of

<table>
<thead>
<tr>
<th>Table W.2</th>
<th>Attributes for the l3FSalesPerson auxiliary class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>l3FRegion</td>
<td>The region that this salesperson &quot;owns&quot; (identifies a sales territory)</td>
</tr>
<tr>
<td>l3FSpecialty</td>
<td>A salesperson's specialized product lines</td>
</tr>
<tr>
<td>l3FTSAWinLoc</td>
<td>Window location preference for TSA (x/y screen start)</td>
</tr>
<tr>
<td>l3FTSAWinSize</td>
<td>Window size preference for TSA (x/y lengths)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table W.3</th>
<th>Attributes for the l3FCustomer class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>l3FCustNo</td>
<td>Customer number</td>
</tr>
<tr>
<td>l3FCustName</td>
<td>Customer name</td>
</tr>
<tr>
<td>l3FSalesperson</td>
<td>Name of the salesperson responsible for this customer (may be more than one)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table W.4</th>
<th>Attributes for the l3FCustomerContact class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>l3FSalesperson</td>
<td>Name of the salesperson who made this contact</td>
</tr>
<tr>
<td>l3FContactData</td>
<td>Free-form text describing the contact</td>
</tr>
<tr>
<td>l3FFollowup</td>
<td>Is a follow-up needed? If so, this optional attribute has a date indicating when.</td>
</tr>
</tbody>
</table>
the class may be stored (the possible superiors attribute, or possSuperiors, in Windows 2000).

The l3FSalesPerson is easy, since it is a user class (remember that it is ideally an auxiliary class) and user already exists in the directory, defined to be contained in Organizational Units. So someone at corporate headquarters already decided where user objects reside in our directory (and if they didn’t, see Chapter 24). What about the other objects?

l3FCustomer—We don’t have thousands of customers, but we may have dozens. How best to organize them? We could organize them by region, by customer type (large, medium, small), by product mix, or by any of many other measures. Since we have so few of them, we’ll just put them in one big customer container of type Organizational Unit and call it L3FCustomersOU. It makes sense to plan ahead for the day when we have thousands of customers and decide how to organize them. We are under intense pressure from Wabash to get this application up and running, and so we will take the easy way out for now, but, we will write the code in such a way that we could develop a better customer organization scheme at a later date (we won’t hard-code any customer organization dependencies).

l3FCustomerContact—These are specific to a salesperson, so we might as well make the possible superiors attribute OU and store them in the same place as the salespersons. We could make the possible superior l3FSalesPerson, but we may need a structural class as a possible superior, instead of an auxiliary class (depends on how your directory handles auxiliary classes). So we chose the same kind of container that contains the L3FSalesPerson objects (OU).

l3FProduct—The products could be organized in as many ways as one could imagine—by market segment, brand lines, or even region (southern Indiana ferrets don’t eat Fois Gras, for example, and so regional product lines exist). However, since we have so few products and no real organizational need among them, we will create one generic container for all of our products called L3FProductOU. The possible superiors for l3FProduct will be organizational unit (OU).

For an overview, see Figure 23.4.
Updating the Schema

On the CD, you will find the complete code for TSA, including code to update the schema. We could use LDIF (LDAP directory interchange format) files to represent the necessary schema changes or we can update the schema with LDAP C code. Some directories provide still other options (such as Windows 2000 Active Directory Services interface scripts or programs).

At the time of this writing, LDIF is still an Internet Draft. This means that while it is mostly defined, it has not yet been given an RFC number. Some later case studies will use LDIF. For this case study, we’ve chosen to add our new attributes and classes to the schema programmatically, using the LDAP C API.

Directory Vendor Differences

You may notice that updating the schema of a directory will have some vendor-specific nuances. For example, one of the nuances for Windows 2000 is that there is a separate in-memory copy of the schema that isn’t automatically updated when you update the schema. To update the in-memory copy immediately, you can set a specific attribute to trigger the update. On any other directory, that particular attribute won’t exist. In fact, another directory may not even have an in-memory schema copy.

Updating the schema is not the only place where directories differ by vendor. Perhaps the biggest difference concerns schemas. There is not (yet) one standardized schema.
that everyone has agreed to implement. There are, however, common elements that most vendors implement, and in the future schemas will converge even more.

**Other Schema Issues**

Just because we have schema classes and attributes defined doesn’t mean we can relax. Here are some other issues concerning the schema that we must also take into account.

- How are we going to apply the schema changes and when? Who has authority to do it? Are there any bad effects on TSA if the changes are not yet made to the schema of every server? In our case, all of our directory servers are replicas, so we probably won’t have too much trouble, but we should make sure that TSA can handle a mismatch in schema updates. For this example, we will assume that the schema changes are made by the schema administrator in Wabash.

- Perhaps TSA should include a schema versioning technique, so that it knows what level of schema it is seeing when it connects to a server. After all, at some point in the future we will need newer versions of TSA and probably new schema additions to support some new features. We could implement such a scheme using a special attribute called 13FTSAVersion, for example, and use the well-known major/minor scheme to detect when versions are compatible and incompatible. We will put this on our “elegance list” for future enhancements.

- Where do we obtain the object identifiers (OIDs) that uniquely define each new attribute or class? For this example, we have generated OIDs for you (and they are only valid for testing—do not re-use them). If you were building this application, you would have to obtain OIDs from ISO, ANSI, someone in your company entrusted with doling them out, or someplace else.

- How do we know the names we have chosen for classes, such as 13FCustomer, are unique? In this case, we prefixed our company name, so we might reasonably conclude that we are okay, but are we sure we have unique names? Both the OIDs and the names need to be unique for all time (at least for all time within our company’s business dealings). Some directory vendors supply recommended naming conventions. Unfortunately, there is no standard. We will assume that the prefix 13F is good enough to ensure uniqueness.

**Directory Structure**

We probably don’t have the freedom to decide the directory infrastructure just because we are creating TSA. The directory was likely constructed for a bigger purpose than just this one application. We will assume that there is a multimaster repli-
cation scheme and numerous, or at least several, directory replicas spread around the L3F company. All of the directory servers can be accessed via the corporate intranet, so no matter where a salesperson is, he or she can find a directory server.

To make TSA more user friendly, we won’t make the salesperson find his or her own directory server: We will keep a local cache of directory servers that can be accessed. (See Figure 23.5.) This local cache for finding directory servers is taken from the locatew example in Chapter 5.

**How TSA Works**

The very first time TSA is ever started on a new machine, it has to contact a directory server. Instead of asking the user for the name of a directory server, we will
automatically fill in a selection box with the LDAP URL that corresponds to the
directory server running at corporate headquarters in Wabash.

LDAP://wabash.L3F.com

The first time TSA runs, it also does some bootstrapping. First, it asks the user to log
in for the first time on this machine. TSA reads the directory information and stores
it locally on the machine for subsequent uses (not unlike the example in Chapter 5).
It then reads the following information.

- The location of all of the known L3F directory servers. This information will be
  used to avoid making the user enter directory server names in the future. Every
time TSA comes up, it reads this information again, in case it changed. How did
the location information for all of L3F’s directory servers get in the directory?
  Our directory servers advertise themselves, like the client and server examples in
  Chapter 9 (TSA is the client and the servers are the directory servers).
- Any relevant L3FSalesPerson information for this user. This will include any
  TSA preferences for this user. For example, this user may have purchased a
  new laptop or visited a new sales office. The user preferences he or she set on
  the last machine he or she used are in the directory. Rather than make the
  user set the application window size and location preferences again, they are
  pulled down from the directory. These are simplistic examples of user prefer-
  ences, but they illustrate the power of centrally stored data for each user.
- Laptop replacement, as mentioned before, is not just for roaming users—it is
  a common scenario even among the nonroaming.
- Any relevant L3FCustomerContact information. We default to reading in
  only contact information for which follow-up is required.
- Read in the product catalog information. This is used for reference when the
  salesperson makes sales calls. All product data is pulled from the directory
every time the user logs in, because even though it doesn’t change often, it
does change. We could optimize this by reading in only product information
whose changed attribute (a new attribute that we would need to define) had a
date later than one that we cached from the last log in. This would be good if
we had thousands of products, but we don’t.

Reading all of this information may take some time, especially since we probably con-
nect to the default server back in Wabash (on first invocation). To be friendly, we will
make directory accesses on separate threads from the user’s GUI thread. We could
have used asynchronous LDAP function calls to avoid blocking. We have to allow TSA
to be responsive to user commands at all times, although some windows may have to
display, “Still Working on that, Susan” (or whatever the salesperson’s name is).

Once we have bootstrapped the new machine, we have achieved steady state. In
steady state, various user actions will cause updates to the directory data, such as cre-
ating a new user contact.
Other than the code, that’s it. TSA shows you how to use directory services to enable roaming users.

Remember that every gun is loaded, and every program has bugs. So don’t point this program at anyone you like. Treat TSA as an example, not a complete solution.

**Write Activity**

Directories are optimized for read-mostly data, but our TSA application does a lot of writing, or so it seems. What kinds of data is it writing and what kind of change activity does that represent?

- User preferences, such as window sizes and locations. These change frequently. We do not, however, send updated directory information every time we notice that a window size or location changed. That would be an unacceptable amount of traffic (a Windows program will receive hundreds of location messages as a window is moved across the screen). Instead of changing these types of settings when they occur, we only change them when (1) the user says to save preferences or (2) the program terminates normally. We are not worried about the time it takes this data to replicate, since the salespeople don’t connect to multiple directories within a short period of time. They are likely to connect to only one directory server at any given branch (and it takes time to drive between branches).

- Customer contact information. A salesperson may call on two or three customers in a day, so the update frequency for contact information is not high. The salesperson does not enter orders using the directory, although he or she will find the order entry service using the directory as a locator service (the order entry service is not implemented in this example).

- Products, salesperson objects, customer objects, and other administrative objects. All of this data is written only as a result of someone at headquarters making updates. For example, new customer objects are created by the order entry program, new salesperson objects are added by an HR application, and products are added by the product manager. These things all represent very low change frequency. In our case, they also happen to represent very low change volumes.

**Write Collisions**

What happens if the same object is modified on more than one directory server at roughly the same time? If that happened, changes could be lost, since there can be only one winner in a change operation (and the replication topology can affect when the collision is even detected). For TSA, write collisions are extremely unlikely. User data is only changed by a single user, who can’t be connected to multiple directory servers at the same time. In fact TSA users are extremely unlikely to connect to two directory
servers within a short period of time—and also make changes that conflict. Administrative data is only changed at corporate headquarters by administrative programs, and we know that authority to run these programs is tightly controlled (well, you know it now).

**What TSA Looks Like**

Figure 23.6 shows a screen capture of TSA's “Follow-ups” dialog box. This dialog box shows the user his or her personalized list of customer contacts that require a follow-up call or visit. This dialog is created when the user selects a menu item from the TSA main window. The list is sorted by date to make it easy to see what should be done next. Although TSA doesn’t implement it, it would be good to provide an audible cue for any follow-ups that are overdue when the user logs on (“ding, you have an overdue follow-up”).

TSA won’t win any user interface guidelines prizes. There are many, many UI features that could make TSA more friendly and pleasant to look at. However, this wasn’t intended to be a UI example and many of those nice features would overly complicate the code.

**Warning!**

On the accompanying CD, you will find the complete code, plus screen captures for all of the windows and dialogs that TSA creates. If you have Windows 2000, you can execute TSA.EXE on the CD and run TSA for yourself. However, TSA updates the
Active Directory schema and creates new directory objects. We highly recommend that you only run this program on a test machine.

Modifying the schema and creating directory objects require a very high authority level. Since this is only recommended to be done on a test machine (preferably a domain controller that is its own Windows 2000 forest), you’ll have no trouble logging on as administrator and then running TSA.EXE.

**Critique of the Example**

The Sales Application (TSA) seems to fit its intended purpose, which is somewhat limited in scope. Here are a few features that would improve it.

- Allow the salespersons to roam outside of the sales offices as well as between sales offices. That is, using a laptop in a hotel room should be just as easy as it is in one of the sales offices. This could be done a number of ways, including installing modems on the sales office servers (and then being extremely careful with security procedures). TSA does not transfer huge amounts of data, so even slow connections should be supportable. If the amount of data for customer contacts were a problem (as for a company with much more contact data), techniques for only downloading some of it could be used (such as intelligent caching).
- Support the TSA application itself as roaming and life-cycle enabled. That is, each time a salesperson connects, TSA should check for a newer version of itself by looking at a TSA profile in the directory. If TSA sees that it is down-level, it should ask the user if he or she wants to upgrade now or later (the profile should also be set up to allow optional upgrades and mandatory upgrades).
- Taken one step further, when a salesperson gets a new machine and attaches it, TSA can be installed automatically. This may require that a generic application agent be written by L3F. This simple and small agent would always have to be installed on each machine, but then it would check a directory resident profile to see which other applications should be installed and take care of installing them. This would enable corporate headquarters, back in Wabash, to control all of the important applications on each salesperson’s machine (and support roaming as before) by controlling the profiles in the directory. This is one reason to have a separate application object in the directory for each application in the L3F company.
- We assumed that we were using TSA over a trusted corporate intranet. What if we wanted to move this application to the Internet? We probably need to examine all the data flowing between TSA and the directory servers and determine if any of it is sensitive. If it is we would have a security problem. We could use LDAP over SSL, for example, to improve our data confidentiality.
PART 4  ■  INTERNET AND INTRANET CASE STUDIES

- You may have noticed that the customer list, the one showing which salesperson is responsible for which customer account, is somewhat more security sensitive than the other data being managed by TSA (even on the trusted corporate intranet). Salespersons should not be allowed to take an account from another salesperson; a regional sales manager or someone of a higher authority should own the customer account list. This means that access control on the customer account objects should be set so that salespeople can read it but can not modify it. Likewise, only people with even higher authority should be allowed to modify the product objects in the directory.

- Suppose a machine is used by many different salespersons. We might not want one salesperson to be able to get sensitive, locally cached information about another salesperson who has used this machine. Even the product catalog might contain sensitive pricing information (in a future enhancement) that we don’t want just any user of this machine to access (we made the user log on, but that may not be enough to stop him or her from peeking at files and the Windows registry, for example). TSA could flush the cache when it notices that it is started on behalf of another user, but that leaves a security hole between the time when the user logs on and when he or she starts TSA. We could encrypt any data that we write into the local cache assuming it gets hardened in a file or a registry. We could keep the cache only in memory for any sensitive data and then that data must be read in whenever TSA starts. We could mandate that the laptops support security features such as Access Control Lists on files and registry keys (such as Windows NT or Windows 2000). Options abound. You will have to choose a solution, or choose to ignore the problem.

- There might be a better bootstrapping method (for the first time TSA is ever run) than asking the user to enter the name of a directory server. Even though we filled in the name automatically for the user, that could still be a problem if the server name or location needs to change in the future. We could have advertised the location of LDAP servers using DNS SRV RR records, for example, and put more smarts into TSA to search for servers on its own. Some directory servers advertise themselves in DNS already.

**Summary**

In this chapter we developed an application, called TSA, to enable roaming users. The application stores and retrieves personalization data from a directory service, so that users can roam among sales offices or use different computers without reapplying all of their personal preferences. The application also stored information that the salespersons need to do their jobs, such as a product catalog and customer contact information. A key requirement our application satisfied is that it supports true roaming from machine to machine and location to location, throughout a decentralized enterprise.
Chapter 24

**CORPORATE EMPLOYEE DIRECTORY**

One of the more common uses of directory services within a company today is to provide an internal corporate directory. Every company that has more than one business location has one. Employees use it regularly to contact individuals across the organization. This chapter uses the concepts from previous chapters to build a schema for a corporate employee directory.

**The Purpose of a Corporate Employee Directory**

Once a company grows beyond a certain size, each employee cannot hope to remember the names of all of the people in the company. Some tool is required to help people find each other across the organization, to call them on the phone, send them a fax, or write them an e-mail message. The purpose of a corporate employee directory is to facilitate communication by helping to locate people in the company. In some organizations, a new employee is really not considered “in the company” until he or she appears in the corporate employee directory.

In many organizations, the corporate employee directory follows the layout of the organization and is the primary means for people within the organization to locate each other. The most common methods of communication between individuals and groups today is through phone calls, pages, e-mail, fax documents, or hard-copy intracompany mail. Less common today, but emergent in the computing industry, is the notion of agents, acting on behalf of the user, interacting with each other to negotiate communication. Any corporate employee directory should be extensible to assist in this more advanced form of communication.
A corporate employee directory consolidates the information necessary to locate people and groups. Separate human resources systems typically contain detailed employment information, and part of this information is useful in locating and contacting them within the company, so a corporate employee directory uses human resources information as its primary information feed.

Other information feeds are required as well. In many organizations, a facilities management group chooses the physical location for the employee, determining when offices should be created, moved, and occupied. Finally, the employee is a primary information source for some of the information in the directory. In many cases the individual is responsible for ensuring the directory’s accuracy for the entries in the directory that represent them.

**Use Cases for the Corporate Directory**

In order to define a usable data model for any directory application, we need to evaluate the usage cases for the information to understand the data content and layout requirements for the information. In this example, the primary use of the directory will be to facilitate communication between employees. The communication methods that are used most are e-mail, phone, and pagers. Other forms of communication such as hard-copy intracompany mail and fax will not be emphasized, although information pertaining to these communication methods will be available in the directory. This directory will be used as follows.

- Employees look up e-mail addresses of other employees in the directory. Lookups are performed by last name, first name, full name, location, and job title of the employees to be contacted.
- Employees look up phone numbers of other employees, using the same search criteria as for e-mail addresses.
- Employees look up pager and fax numbers of other employees, using the same search criteria as for e-mail addresses.
- Employees look up the chain-of-command for other employees to establish the organization and management chain that another employee is under.
- Employees manage the information about themselves that is contained in the directory.
- Human Resources adds, updates, and deletes information about employees in the directory as changes are applied to the records that correspond to the employees.
- Employees build distribution lists for sets of e-mail addresses and use these in sending groups of people e-mail messages.

With these use cases we can develop a data model for the information.
Data Model

The use cases discussed in the last section dictate the data model requirements. The types of searches required indicate the set of attribute types that must be defined to support the search operations. The uses of the information returned indicate additional attribute types that must be defined.

Object Classes

The use cases for the information dictate the set of entities that exist for this model. Specifically, the use cases show that the following object classes are required.

- employee
- department
- distribution list

These entities are related to each other to form a data model as shown in Figure 24.1.

Notice that in the model, departments are related to other departments by a containment relationship. This relationship is implemented using the hierarchical structure of the directory. Each department has a manager—an employee that is so defined in the directory. Some employees have administrative assistants. For those who do not have an assistant explicitly noted, their assistant is defined to be the assistant of the first manager up the chain of command (department hierarchy) who has an assistant.

Distribution lists contain three multivalued attributes, one for the “to:” list, one for the “cc:” list, and one for the “bcc:” (blind-copy) list. These three attributes contain the names of employees that who to receive messages sent to the distribution list.

![Diagram of the data model for the corporate directory](image-url)
Attribute Types

By looking at the use cases for this information, we can see that the following attribute types must be defined in the model. These attributes are tied to the object classes outlined in the previous section. For the employee entity the following attributes are defined.

- last name (surname)
- first name
- full name (commonName)
- location
- job title
- phone number (telephoneNumber)
- fax number
- pager number
- e-mail address
- physical address
- department
- employee serial number
- assistant
- manager

Some of these attributes are required; others are optional. Attributes used in naming the directory entry and information that always exist for a person are noted as required. Other information in the employee object class is listed as optional. For the department entity, the following attributes are defined.

- description
- manager

For the distribution list entity, the following attributes are defined.

- name
- description
- “to:” list (member)
- “cc:” list
- “bcc:” list

The LDAP schema that implements the data model is defined in Code Example 24.1 as a package. (Description and the use of packages for defining schema are discussed in Chapter 17.)
Name Space Layout

The name space layout for the corporate directory should follow the hierarchy of the organization. Using this for the hierarchy makes the directory easier to understand for employees who will be using it. This layout also helps determine the chain of command across the organization.
Figure 24.2 shows the name space layout for the corporate directory. Employees reside within departments. Departments follow the organizational structure of the organization. At the top of the directory tree is an entry representing the organization. This entry should have a distinguished name that makes this tree of information usable in a global name space. The name of the organization is usually the best choice for this top-level entry. All information that pertains to an organization and resides in directories that the organization manages should exist below this top-level entry in the directory.

Now that the object classes and attributes are defined for the corporate directory, the directory schema for these entities can be deployed in the directory servers that will be used. Deploying this schema depends on the directory implementation(s) that is (are) used in the organization. Some directory servers allow dynamic updates to the schema using the LDAP modify operation to add `attributeTypes` and `objectClasses` attributes to the directory’s subschema subentry in the name space. Some servers, on the other hand, require updates and additions of additional entries in the directory, which represent the directory schema. Still others require an off-line update to the configuration of the directory server, resulting in a restart of the directory server. After this step is completed, the attribute types and object classes will be available to programs updating and accessing information in the directory.
Accessing the Corporate Directory

Using the directory schema, we can describe the set of operations to be performed on the corporate directory in terms of LDAP operations against the schema. Refer to the example for this chapter on the CD to see these parameters expressed in calls to the LDAP client in order to return the information.

Looking Up People

Looking up people in the corporate directory involves performing an LDAP search operation against the directory. The search operation allows a single entry, list of entries one level below an entry, or whole subtree of entries below (and including) an entry to be searched. Subtree searches are the most common methods of finding information when all that is known about the person is a last name or phone number. One-level searches are useful in determining the members of a department. Single-entry searches are useful in discovering all attributes about a person, department, or distribution list.

For these examples, the root of the name space (that is, the distinguished name of the organization entry for this example) will be assumed to be “o=widgets”. In order to find a person in the organization, given their last name, perform the search described in Code Example 24.2.

In order to find a person in the organization given just the last four digits of his or her phone number, perform the search described in Code Example 24.3.

Finding the set of employees in a department, given the department’s name, requires two searches: one for the full name of the department, then a one-level search of the department. Perform the searches described in Code Example 24.4.

| search base: o=widgets |
| search scope: sub-tree |
| search attributes to return: cn, telephoneNumber, mail, jobTitle |
| search filter: (&(objectclass=employee)(|(sn=<last name>)(cn=*<last name>*))) |
| perform the LDAP search operation |

**Code Example X.2** Finding a person by last name
Looking Up a Chain of Command

The looking up of the chain of command in the directory involves returning the names of all managers of all departments between a given employee's name and the top of the organization. This will involve a number of base searches of the directory, starting with a search to determine the full name of the employee. Code Example 24.5 shows the algorithm, presented in pseudocode.

The algorithm in Code Example 24.5 will display the chain of command and bottom-up order (the employee at the top of the list with the highest manager in the chain of command at the bottom). To present this information in a different order, use a stack to store the information as it is returned, then display the entries when removing them from the stack.

```plaintext
search base: o=widgets
search scope: sub-tree
search attributes to return: cn, telephoneNumber, mail, jobTitle
search filter: (&(objectClass=employee)
    (telephoneNumber=*<four digit extension>))
perform the LDAP search operation

Code Example X.3  Finding a person by phone extension
```

```plaintext
search 1 base: o=widgets
search 1 scope: sub-tree
search 1 attributes to return: cn
search 1 filter: (objectClass=department)
perform the LDAP search operation
search 2 base: <full department name from search 1>
search 2 scope: one-level
search 2 attributes to return: cn, telephoneNumber, mail, jobTitle
search 2 filter: (objectClass=employee)
perform the LDAP search operation

Code Example X.4  Finding all persons in a given department
```
Adding a Distribution List

Adding a distribution list is our first example of a modification to the content of the directory. The intent of distribution lists is to provide a way of grouping sets of people and departments together to assist in addressing electronic mail. Distribution lists can be created by individuals in the organization and, because these distribution lists are stored in the directory, can be used by others in the organization. The directory provides a convenient sharing mechanism for these distribution lists. Code Example 24.6 provides pseudocode for adding a distribution list to the directory, given the person’s name, the name of the distribution list, its description, and the names of the members of the list. Note that the members of the distribution list can be either other distribution lists or individuals.

```
// Find the distinguished name of the employee:
search base: o=widgets
search scope: sub-tree
search attributes to return: cn
search filter: (&(objectclass=employee)(sn=<last name>)
  (cn=*<last name>*)))
perform the LDAP search operation
// Now use each component of the distinguished name
// to perform a search to find the department's manager's name:
while the remaining name is not "o=widgets" do
  remove the first relative distinguished name from the
  remaining name
  search base: <distinguished name less the first relative
  distinguished name>
  search scope: base
  search attributes to return: manager
  search filter: (objectclass=department)
  perform the LDAP search operation
// Now use the manager's name to look up the manager:
search base: <manager's name>
search scope: base
search attributes to return: cn, telephoneNumber, mail,
  jobTitle
search filter: (objectclass=employee)
perform the LDAP search operation
```

**Code Example X.5** Finding the chain of command for a person in the organization
Modifying a Distribution List

In order to add or delete a member of the distribution list we must modify the distribution list entry. Code Example 24.7 gives pseudocode for making a modification to the distribution list. While multiple modifications can be performed by the LDAP modify operation, this pseudocode assumes that only one modification is made at a time.

Deleting a Distribution List

In order to delete a distribution list from the directory, we only need the distinguished name of the entry. The pseudocode for this operation is shown in Code Example 24.8.
Managing Information in the Corporate Directory

The examples shown to this point dealt with operations that employees in the company would be performing against the directory. While addition, modification, and
deletion of distribution lists are updates made to the directory, a different set of operations is required for managing the rest of the directory. These operations manage the structure of the directory tree, including the addition, modification, and deletion of departments and persons in the directory. More complicated operations are necessary when a person transfers or a reorganization occurs.

Adding Employee Information

Adding a new person to the corporate directory is a relatively easy operation. This involves setting up for and invoking the LDAP add operation. The entry is added to the directory using the person’s name and department. Code Example 24.9 gives the pseudocode for performing this operation.

Modifying Employee Information

A corporate directory needs to reflect all changes in employee title, location, mail address, telephone number, and so forth. These changes are performed using an LDAP modify operation. Code Example 24.10 shows the pseudocode for performing this operation.

Deleting Employee Information

When a person leaves the organization, his or her entry must be removed from the corporate directory. Updates to any departments or employees to whom the person reported or supervised must also be updated. Code Example 24.11 provides the pseudocode for these operations.
Handling Employee Transfers

When a person transfers from one department to another the corporate directory must be updated to reflect this change. If distribution lists exist below this person’s

```
// First find the distinguished name of the Department
search base: "o=widgets"
search scope: sub-tree
search attributes to return: cn
search filter: (&(objectclass=department)(cn=<department name>))
perform the LDAP search operation
// Using the distinguished name of the entry returned in the first search
// add the new employee entry
distinguished name for add:
  cn=<name of person>, <name of department>
attribute 1: cn=<name of person>
attribute 2: objectclass=employee
attribute 2: objectclass=organizationalperson
attribute 2: objectclass=person
attribute 2: objectclass=top
attribute 3: sn=<last name of person>
attribute 4: telephoneNumber=<person's phone number>
attribute 5: mail=<person's e-mail address>
attribute 6: employeeSerialNumber=<person's company serial number>
perform the LDAP add operation
```

**Code Example X.9** Adding employee information to the corporate directory

```
// Find the distinguished name of the employee:
search base: o=widgets
search scope: sub-tree
search attributes to return: cn
search filter: (&(objectclass=employee)
  (|(sn=<last name>)(cn=*<last name>*)))
perform the LDAP search operation
// Now use the distinguished name to perform the modify
attribute modification: REPLACE jobTitle=<new title>
attribute modification: ADD/DELETE mail=<e-mail address>
distinguished name for modify: <name of person>
perform the LDAP modify operation
```

**Code Example X.10** Modifying employee information in the corporate directory
entry in the directory, a simple modify name operation cannot be performed; instead the person’s entry must be copied to the new location and all the distribution list entries below the person’s entry must be copied under the new entry in the directory. After the new entry is created for the person, all existing distribution lists that con-
tain the person's name must be updated with the new name. As a final step, the old entries for the person and distribution lists must be deleted from the directory. Code Example 24.12 shows the pseudocode for these changes.

```plaintext
// Find the distinguished name of the employee:
search base: o=widgets
search scope: sub-tree
search attributes to return: <all>
search filter: (&(objectclass=employee)
  (|(sn=<last name>)(cn=*<last name>*)))
perform the LDAP search operation

// Now use the returned information to create a new entry in the employee's new department
distinguished name to add: cn=<name of person>,
  <new department distinguished name>
use all attributes returned from first search
perform the LDAP add operation

// For each distribution list below the old employee entry, move it to the new entry
search base: <old name for employee>
search scope: one-level
search attributes to return: <all>
search filter: (objectclass=*)
perform the LDAP search operation
for each returned entry do
  new distinguished name: cn=<returned common name>,
    <new name for employee>
  attributes to add: <all attributes returned>
  perform the LDAP add operation
od

// Update all existing distribution lists to have the new name for the person
search base: "o=widgets"
search scope: sub-tree
search attributes to return: member, ccList, bccList
search filter: &(objectclass=distributionList)
  (|(member=<old name of person>)
   (ccList=<old name of person>)
   (bccList=<old name of person>))
perform the LDAP search operation
for each returned entry do
  name of entry to modify: <name returned from search>
  find the <old name of person> in the member, ccList, and/or bccList attributes

Code Example X.12  Transferring an employee to a new department  (continued on next page)
Handling Company Reorganization

When a company reorganizes, the department names and structure may change. This very large number of updates to the corporate directory can be viewed as generalizations of the steps shown in the preceding examples. Indeed, if the top-level department name changes, or one company merges with another, the distinguished names of all entries below these top-level entries must be changed to reflect this update. If the directory service implementation does not support the modify name LDAP operation, it may be necessary to unload the directory information and perform an off-line update to the directory entries to effect the changes. For moderate to large directory name spaces, this will be much more efficient than recursively copying every entry into the new name and then deleting the subtree of old entries after the copy is complete. Furthermore, because entries contain distinguished names of other entries as values of attributes within the entry, a modify name operation will not update these names. Additional operations would be required to update these distinguished name attribute values once the new names are defined.

Sample Code

The code has been compiled and tested on an OS/390 system using the OS/390 Security Server LDAP Server for testing both the information model and update/access routines. Command-line programs that perform each of the basic operations are provided allowing scripts to be written around the basic operations.
While it is not directly used by either the LDAP server or client, keeping track of the LDAP schema used by the corporate directory is useful for continued management of the information structure in the directory. The package model described in Chapter 17 is used here to provide a means of documenting additions to the schema that are specific to the corporate directory. The LDIF format of this package entry (except for the distinguished name of the package entry) is provided on the CD for use in building this example.

Source code that implements the functions described in this chapter can also be found on the CD in the directory specific to this chapter. The source code, written in Java, and makes calls to access an LDAP directory using a JNDI LDAP service provider.

**Using the Corporate Directory from E-Mail Systems**

Many electronic mail systems extract the mail attribute for holding the e-mail addresses of employees and place it directly on the “to:”, “cc:”, or “bcc:” lists when composing e-mail messages. This provides an easy, inexpensive mechanism for tying together an organization’s e-mail system with its corporate directory.

In addition to handling individual employee definitions, many mail systems also make use of the groupOfNames object class for building distribution lists. Distribution lists are evaluated using the member attribute; each attribute value is used to build the “to:” list. If the entry is found to be another groupOfNames entry, the member attribute is read again. This continues until all nested groups have been evaluated and a list of recipients is built in the mail message being composed.

**Critique of the Solution**

While the solution points out many of the more complicated aspects of managing a corporate employee directory, it does not address a number of issues. Another approach was described in Chapter 3 using a Web-based CGI program. (Refer to Chapter 3 for the limitations of that approach.)

The solution does not cover replication of the directory information across multiple servers. Replication is used to provide better scalability as well as reliability.

Nor does the solution deal with security issues revolving around access to and update of information contained in the directory. With the types of information stored in the directory for employees and distribution lists, access controls should be imple-
mented to restrict update access to only those people and applications that require it. A number of directory servers support access control mechanisms that allow specification of access control in a hierarchical fashion. This allows whole subtrees of information to be controlled using a single access control specification. As an example, the distribution lists held underneath an individual’s entry should be updateable by the individual entry under which they appear as well as by the application responsible for accomplishing the transfer of personnel between departments.

The solution does not discuss automated updates to the employee directory that might be required to synchronize the information in the directory and the information stored in other, pre-existing directories that are found in the enterprise. Metadirectory solutions can be used to provide this synchronization service. Other automated mechanisms, such as background agents, could be used to manage the directory content, verifying that dn-pointers are still valid and noting discrepancies.

Where updates to multiple entries in the directory are required, the solution does not provide retry logic or checkpoint processing in order to resume a request from where a previous, failed attempt left off. Instead, the solution is written to allow retry in order to clean up a failed request.

Summary

This case study has shown how to extend the object classes in the basic LDAP schema to include attributes that are specific to the organization. A name space layout was created that uses department and distribution list entries to group sets of employees together. Further, a relationship was created using assistant and manager object classes. The case study is implemented using Java and a JNDI LDAP service provider. Full source code is available on the CD.
Custimized Web sites have become essential for e-Business. When people interact with a company on the Internet, they expect the same level of customer service they might get when dealing with a person. Sometimes they don’t get that level of service on very complicated Web sites. Some companies have more than one Web site, each built and maintained by different people. One Web site may know you and another may not, even though both represent the same company. For example, one site may know your preferences and may ask you to log on. Another site may either not know you or may ask for a different log on sequence; or it may seem constructed to intentionally annoy you by forgetting that you have a request outstanding (the Web site doesn’t know about the order-entry site). Most Web sites are not constructed to annoy you; instead, some were just constructed without the benefit and exploitation of a good directory service.

The Customer Access and Update

The Little Fuzzy Ferret Friends Company (L3F) introduced in Chapter 22 is considering letting its customers access information on their Web site. Several large shipments of Weasel Wieners spoiled due to out-of-date shipping addresses. This and other recent customer problems have been traced to out-of-date customer contact information. The large growth in customers and sales have made it increasingly hard for sales employees to maintain current customer information in the company’s directory. In addition, territory reassignments make it hard for customers to contact their assigned salesperson. The L3F Company has decided to use the Web for customer self-service in these two areas.

This is a major step beyond publishing “brochures” on the Web. The company has decided to allow customers to update their own customer contact information in the directory and look up contact information for their salesperson. Since customer information is confidential, we will also need to consider the security of the directory data.
The Web Self-Help (WSH) Site

The Web Self-Help (WSH) site will start small, with three major pages.

Home—The home page is where customers log on.

Sign on—This page, while hidden from the customer’s view, does the work of getting customers signed on.

Customer profile—This page displays the customer information, including shipping address and salesperson contact information. This is where the customer has the opportunity to change the shipping address.

The user IDs are currently managed outside the WSH application. Each customer will need to be given a directory user ID. (For this sample the user IDs are defined in Microsoft Active Directory.) This user ID is then stored in an attribute associated with the customer account. A self-enrollment application can be created to reduce the administrative burden associated with this.

The customer profile page uses several operations to retrieve all the requested data, including searches to find the salesperson contact information. Modifying the shipping address changes a stored attribute.

Schema

Instead of building a schema from scratch, we will expand the schema from the roaming example in Chapter 23. In that example, we created an initial directory infrastructure representing salespeople and customers. We can now use several of the directory objects defined during that project to facilitate the sharing of information across applications. (See Table 25.1.)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13FCustomer</td>
<td>Contains customer information. Attribute for shipping address and logon ID will be added.</td>
</tr>
<tr>
<td>13FProduct</td>
<td>Contains product information. Will be used unchanged.</td>
</tr>
<tr>
<td>13FSalesperson</td>
<td>Contains salesperson information. External contact information will be added.</td>
</tr>
<tr>
<td>user</td>
<td>The Active Directory class for a user.</td>
</tr>
</tbody>
</table>
The WSH application will expand the l3FCustomer class to include the shipping address. Customers, who log on using their registered user ID (l3FWSHCustomerUID attribute), will be allowed to read their complete record and modify the shipping address (l3FShippingAddr attribute). Customers will also be shown the contact information for their salesperson (see Table 25.2.)

### Directory Infrastructure

Our biggest directory infrastructure concern is where the directory service used by the Web application should be located. Assuming that our Web server is located in a DMZ, as shown in Figure 25.1, we have a choice of placing the directory either inside the DMZ or on the internal network. Given that the directory contains a significant

![Diagram](image.png)

**Figure Y.1** Possible directory infrastructure for WSH application

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>l3FCustNo</td>
<td>Customer number (existing)</td>
</tr>
<tr>
<td>l3FCustName</td>
<td>Customer name (existing)</td>
</tr>
<tr>
<td>l3FSalespersonName</td>
<td>Name of the salesperson responsible for this customer (may be more than one)</td>
</tr>
<tr>
<td>l3FShippingAddr</td>
<td>Customer shipping address (new)</td>
</tr>
<tr>
<td>l3FWSHCustomerUID</td>
<td>The logon ID used by the customer. This matches an ID defined by the security subsystem. (new)</td>
</tr>
</tbody>
</table>
amount of critical business data, we want to limit its exposure if our Web server is broken into. We really have a limited number of ways to do this.

If the directory server is located in the DMZ, we can get some data protection by making it a partial replica of the enterprise’s directory data. One way of doing this would be to use metadirectory technology to replicate just the objects of interest. In this case that would be the customer, product, and salesperson object classes. In addition, we actually only need to replicate a portion of the salesperson objects. (Customers are not allowed to see all of the attributes of a salesperson.) For customers, we can maintain their user IDs only within the directory servicing the DMZ. Since we are using Microsoft Active Directory, these user IDs are also usable for accessing file system shares and other operating system resources. Isolating them to a separate partition of the name space will prevent them from being used in L3F’s intranet by accident. In addition, the metadirectory data can help ensure that the only attribute value change reflected into the master directory is to the customer shipping address.

If the server is located in L3F’s intranet, our real concern is that the Web server has sufficient authority to create new user accounts. In this case we should move the user account creation to a third tier in order to reduce the risk from intrusion. A related problem with giving access to an internal directory service is protecting the other data from direct Web access.

A composite option is to locate a partial replica on the intranet. (See Figure 25.2.) This has the advantages of limiting the damage of a server intrusion in the DMZ, as well as reducing the data that is potentially exposed to Web users.

---

**Figure Y.2** Use of read-only partial replica
The composite option is the best. While it has the disadvantage of running three directory service instances, it provides the best balance of performance and security. The DMZ Web replica can service most user accesses, including user log on. The intranet Web replica provides an additional level of isolation and allows the firewall to audit the directory modification traffic. Finally, all DMZ systems are kept isolated from the corporate directory that contains most of the enterprise’s data. Most directory services require the use of metadirectory technologies to coordinate the Web and corporate directories.

How the Web Self-Help Application Works

The Web Self-Help (WSH) application works by storing customer data in the directory. The data is retrieved after logon and the user ID is used to locate the correct customer object.

First customers type in the company’s URL (currently http://localhost/wsh/home.htm). The page they receive, shown in Figure 25.3, allows them to enter their user ID and password.

![Welcome to the L3F Web site](image_url)

Figure Y.3  The L3F home page
Once they click the button to enter the secured area, a “hidden” page, signon.asp, is given control. This page collects the user ID and password and uses an ActiveX DLL (wshcust.dll) to validate the ID and retrieve the customer’s data. The main asp code is

```asp
' Get the wshcust object.
Set wshcust = Server.CreateObject("wshcust.l3FCust")
if (isobject(wshcust))=false) then
    session("wshlogonreason") = "Failed in creating wshcust object"
    Response.Redirect "badsignon.asp"
end if
' Set the userid and password properties.
wshcust.UserID = UserId
wshcust.Password = Password
' Check the password and populate the properties
r = wshcust.ValidateIDPassword()
```

The second line loads the control. Then the `UserID` and `Password` properties are set. Once this is done, `ValidateIDPassword()` is used to verify the user’s identity. If the verification is successful, control is passed to page `CustomerProfile.asp`. If unsuccessful, the `badsignon.asp` is displayed to give feedback to the customer.

`CustomerProfile.asp` uses the same control to interface with the directory, except that it calls `RetrieveCustData()` instead of `ValidateIDPassword()`, the main difference being that `RetrieveCustData` starts from a DN instead of searching using the user ID:

```asp
' Set up and populate wshcust object
set wshcust = server.CreateObject("wshcust.l3FCust")
wshcust.UserID=custUserID
wshcust.Password=password
wshcust.dn=dn
r = wshcust.RetrieveCustData()
```

After the data is retrieved, it is displayed. The page a customer sees is shown in Figure 25.4.

Now let’s look under the cover a little bit. The workhorse is dll wshcust. It encapsulates all authentication and directory operations. First `OpenDSObject` is used to get the default naming context in order to force the Active Directory to verify the user ID and password.

```asp
Set rootdn = dso2.OpenDSObject("LDAP://" & dNC, wshUserID, wshPassword, 0)
```
After deciding that the user ID is authentic, wshcust searches the directory using an active data object (ADO) search. The code for the search is shown in Figure 25.5.

If the user ID is found associated with a customer record, the DN is saved and the browser is redirected to CustomerProfile.asp. CustomerProfile.asp calls the RetrieveCustData function to populate the ActiveX control, then displays the attribute values from the customer's directory entry. The RetrieveCustData function also does an ADO search to retrieve the salesperson's current contact information from the salesperson's object.
Critique of the Solution

The Web site needs some changes before it is enterprise-ready. First, we currently do not use SSL to protect the user’s password. We should add SSL protection to both the browser-to-Web server data flows and to the Web server-to-directory data flows, in order to encapsulate and restrict display of the customer’s related data. Each page re-authenticates to the directory before retrieving data, ensuring that data is only returned to authorized people. Additionally, customers do not have direct control over the data requested and displayed—they cannot enter another company’s DN.

Second, we need to make a few changes for scalability and performance. As we add customers, the time required for the current search at logon will increase. This is because we are searching the objectclass attribute, which is not indexed in Active Directory. This will affect both the performance seen by customers and the overall application’s ability to scale. The fix for this is to add the objectcategory attribute to the L3FCustomer class and use it instead of objectclass for the initial search. Third, the example is inefficient. In several places we retrieve all of a customer’s attributes. Later, as we add attributes, this wasteful retrieving of unused data will cause the performance of the implementation to degrade. The solution is to expand the functionality of RetrieveCustData so it can retrieve only the attributes that will be needed.
A big limitation of the WSH application is that the pages and code are static. As attributes are added to the directory, wshcust.dll will have to be updated, redistributed, and installed on our Web servers. A better design would allow the application to specify the required attributes and have wshcust not play an active role in translating the I3FCustomer schema. It would also be better to factor the search for salesperson contact information into a separate dll.

Summary

We have provided a sample program showing how a middle-tier Web server can use a directory to personalize an individual’s experience at the site. This shows that a complex user experience, such as updating one’s own information, can be easily provided.
Chapter 26
APPLICATION MANAGEMENT

The only thing constant in life is change.
—FRANCOIS DE LA ROCHEFOUCALD

Application management used to mean simply ensuring there was enough space to install an application. Today application management deals with these issues.

- Carrying out software installation on dozens of servers or thousands of workstations without physically visiting each
- Managing the rollout of new releases or maintenance
- Providing locks or granular permission to use certain features
- Preventing unauthorized changes to the application configuration
- Understanding the state of server applications, such as whether they are alive and operating well. This may touch on operations management.

Let us explore an example of application management using directory services. Some of the principles learned in Chapter 13 will be applied here.

The Application Management Problem

Let’s use a simple client-server application that may be run in either a two-tier configuration or as a three-tier browser-based application. Later, we may see that the principles we cover can extend to a real enterprise-wide application.

Our learning exercise will use a quotation application as its vehicle. Our quotation server contains an internal repository of quotations that represents our database. The implementation was written using Windows 2000, but the tiny quotation server itself will run on any PC that supports the Win32 API. (See Figure 26.1.)

The server implementation is based on a thread-pooled boss-worker design pattern. Server initialization code creates a pool of worker threads. Each worker blocks, waiting for work. The server boss thread listens on a socket port for a client connection. When
a connection occurs, the connection socket is set into a control block for the first idle thread and the thread is unblocked. The thread gets a random quotation from its repository and sends it through the connection socket to the client requestor. The client sends either a request for another quotation or a quit request. Meanwhile, other workers could be getting and processing work from the boss in the same manner. Telnet makes a quick temporary test client for this server. This is not a good long-term client because it isn’t a centrally managed part of our distributed quotation server.

The Purpose of the Application Management

We will remotely manage binding information for use by prospective clients. Binding information often consists of two things.

- The IP address of a service.
- The port used to listen for client requests.
This nugget of binding information dynamically advertises a service to clients. It also enables that service to pick up dynamic configuration parameters—in this case the port at which it should listen for client requests. This binding information could be extended to any number of servers by making it multivalued or by using a naming scheme based on the enterprise organization, geography, or network topology. Security control could be applied to manage access to service locator information based on client permissions. The binding information relates to the first three application management objectives.

We also want to centrally administer other operating parameters of the server.

- A file path for the server exception log, local to the server
- The number of worker threads in the server's thread pool
- A server restart delay for use after a server process aborts

These parameters enable further dynamic configuration for the service. The log location may be used to find the log so as to monitor the health of the service. The restart delay is used to allow a failing service to get past a possible transitory disruption without burning all its retries at once. Configuration directory information may be extended to many servers in the same manner as the binding information. Modification of service management objects in the directory should be restricted in the directory service to administrators only.

These prototypical management points of our application touch on the five areas of application management proposed at the start of this chapter. We have the first hints at how a Web farm could be managed. (See Chapter 28.)

**The Directory**

A plain old Telnet client (POTC) isn’t directory-enabled, so the tester must provide the client service binding information. Both the client and the server sides of our quotation application are directory-enabled. We implement some of the principles discussed in Chapter 13. In particular, the directory-enabled client has these features.

The client will find the quotation service’s DNS name through the directory. The DNS name or IP address equates to the location of the server. Multiple servers may be registered on separate machines by having distinct registration names in the directory. The client chooses a given server by looking for a registered service name in the directory.

The client will determine the quotation service’s TCP socket port number through an attribute in the service name in the directory. Different quotation service instances or versions may be registered for the same server machine but with
different port numbers. The client could choose one by its registered service name.

The directory will do the following for our directory-enabled quotation server.

- The server determines how many worker threads it should start by consulting a directory entry.
- The local log file location is specified by a directory entry. The server logs exceptions and interesting events to this local file after finding its path in the directory.
- The server boss thread uses a directory entry to determine which TCP port to use to listen for client requests.
- If the server must abort, it should do so only if it can fail an individual client’s request and still carry on with the other requests. If failing individual requests isn’t an option, it should attempt to restart after a delay interval, obtained from a directory entry.

Notice that these items correspond to the management items discussed earlier.

**Directory-Enabling the Server**

Let us stipulate that the quotation server is a legacy server. *Legacy* is the term used to describe any piece of software that was written before this week. Our server is ignorant of the concept of enterprise directory. We want to parameterize the characteristics listed earlier to this application. We don’t want to open the source code of the server because it could be a vendor product.

Our server gets its thread pool size and request port number as start-up parameters. It logs interesting events and exceptions to stdout. It takes a “q” console keystroke as a shutdown (quit) request.

We successfully avoided opening the server source code by writing a directory-enabled launcher application that obtains operating parameters from a directory service. (See Figure 26.2.) It starts the quotation server as a child process. The location of the event log, the request port number and the pool size of worker threads are obtained from the directory and administered by a central administrator.

Our server uses stdout for its log file. The launcher redirects stdout to a log file path named in the directory. It specifies that the child process inherits the stdout file handle of the unmodified quotation server. (See Figure 26.3.) Each log item is time-stamped. An item has a worker thread identifier if it is emitted by a worker thread. A number identifies each quotation delivered. Error information is shown when it is encountered, as shown in Figure 26.4.
Legacy server:
• Close source code.
• Take port number as initial parameter.

Launcher (new code):
While not stop command
• Get published quotation connection info from LDAP.
• Spawn quotation server with advertised port.
• Listen for quotation server to abort.
• Wait for published restart interval.

Figure Z.2 Directory-enabled quotation server via launcher

Launcher watching quoter...
20:54:36 'q' ends server
20:54:36 Boss running
20:54:36 Worker 1576 started
20:54:36 Worker 1940 started
20:54:36 Worker 580 started
20:54:36 Worker 2056 started
20:55:36 Dispatched 1576
20:55:36 QUOTESRV: Number of available quotes is 362
20:55:36 TID 1576: Q: 41
20:55:36 TID 1576: quit command
20:55:53 Dispatched 1576
20:55:53 TID 1576: quit command
20:55:55 Dispatched 1576
20:55:55 TID 1576: Q: 180
20:55:55 TID 1576: quit command
20:55:56 Dispatched 1576
20:55:56 TID 1576: Q: 74
20:55:56 TID 1576: quit command
20:56:09 Dispatched 1576
20:56:09 TID 1576: Q: 345
20:56:09 TID 1576: quit command

Figure 26.3 Quotation event log sample output
We use a general, illustrative portable logging mechanism “to record interesting events.” We simply log to a file path obtained from the directory.

**Directory-Enabling the Client**

Our simple client is directory enabled. It is a WIN32 API console application named *quotee*. It uses LDAP to access the service binding information, so it may run on any Windows platform having LDAP or be ported to Linux or any platform where LDAP is available. The client finds the desired service name in the Active Directory, obtains the DNS name of the server, binds to the service port, sends a request, receives a quotation, displays it on the console, sends a disconnect command, and exits. This is shown in Figure 26.5.

The Win32 client could be moved into a three-tier Web application by executing it as a CGI application. The output to stdout would benefit by having some HTML tags emitted surrounding the quotation. Another option would be to write a browser-side Java applet that connects to the server and displays its output on the Web page. The Java sandbox security model dictates that the quotation server must be on the same machine as the Web server unless there is an administrative policy override.

How do the clients or the server locate the directory? Isn’t a good directory supposed to mitigate client-side need-to-know locations of things? Some systems approach this problem in this manner: If the LDAP BIND API doesn’t receive a specific host
parameter, it binds to the nearest copy of the directory. Behind the scene, dynamic DNS SRV records maintain this information.

The quotation application is contained on the CD.

**Schema**

We’ve been glib in implying that we simply use LDAP and read or write this or that directory entry. When we need to put service binding and administration information into a directory we must focus on the schema.

The quotation server is directory-enabled for use on a Windows server. We utilized an existing service connection point in the schema to avoid adding schema changes.

We decided to use a `serviceBindingInformation` multivalued attribute as the container of everything about the quotation service except its DNS name, to keep the sample program short. For that we used a `serviceDNSName` attribute. We’re mixing service administration data with client binding information within the binding attribute for simplicity. This lack of separation may not be appropriate for an enterprise application. It is more secure for administrative information to be invisible to clients.

Our schema has a `serviceAdministrationPoint` object class. Administrative programs for services would use it. Our core quotation server is legacy, that is, directory-ignorant. Only the launcher knows about the directory. The server cannot be administered while it is running. A real service may leverage the directory for real-time administration.

We need an object within the directory to contain our `serviceConnectionPoint` instances. We choose to create a top-level organizational unit called `Published-Services`. (See Code Example 26.1.) It is populated with `serviceConnectionPoint` instances. Each of these has a multivalued `serviceBindingInformation` attribute and a single-valued `serviceDNSName` attribute. The former contains the TCP port number, event log path, and number of worker threads. A client needs only the port number: It gets the DNS name from the second attribute.
Critique of the Solution

Our distributed service exhibits some of the principles that we discussed in Chapter 13. We get some things cheaply, such as being able permit only administrators to modify our service information in the directory. The quotation service exhibits some of the compromises of a learning tool, but a few things are missing.

Our philosophy is that if we know what is missing and what could be a new feature’s added advantage, then we may provide it in a real application. We are missing the following.

A robust remote administration application. We have only a batch command that sets binding and operational information into directory attributes. Our server does not respond to changes while it is running. This real problem could be overcome by providing it with an event handler that listens to events published by more sophisticated administrative code.
Separation of binding information and administration information. We didn’t do this in the interest of tutorial brevity.

Rotation of event log files. Only one log file is used at a time, which means it must be copied and reset while the server is stopped. We should dynamically rotate the log files.

The ability to inspect the log while the server is running. This impacts reliability and serviceability. Logging facilities in many operating systems allow dynamic inspection.

Event analysis. Our only tool is a text file browser; an analysis tool would be beneficial. A daemon script looking for trouble and e-mailing alerts would be a slick project for extra credit.

Dynamic thread pool size. Our thread pool does not expand or shrink based on heuristics.

Dynamic load balancing across servers. We only provide static load balancing. There are vendor products that aggregate servers into farms of servers.

Generalized directory exploitation. As mentioned before, all the service and administration information is in one, simple, multivalued binding.

Server version control. We may change server versions without disrupting client operations, but the version of the server is not recorded in the directory.

Client version control and deployment. Our client may be deployed to multitudes of users through a file share or a Web page. This provides some measure of centralized version control and deployment. A human does not need to visit each client machine, although the service must be advertised to users somehow. Perhaps an e-mail broadcast to the users advertising the URL or file share would be a baby step.

Granular permission to use features is not provided. We have only one feature: “request a quote.” We could provide more features, such as “request quotation number 10.” By enabling the server for administrative policy we could enable users for one or both features. A generic approach would be to place user groups into the operational information for the server in the directory. The client would supply credentials as part of an authentication exchange with the server. The server would grant access to a feature based upon authenticated credentials matched against security groups registered for the feature.

By directory-enabling our application, we gained points in the areas of reliability, availability, serviceability, scalability, and security. We didn’t get an A on all of these principles, however. Let’s go over the limitations of our approach so we’ll know how to improve when we implement a real application.
Reliability

We cannot inspect the log while the server is running so we cannot know about
minor problems on a real-time basis. We need this information to ensure that the
server is running reliably.

We have generalized some principles that could be applied across platforms such as
Unix, MVS, Windows, or AS/400. Our roll-your-own event log could be extended
into something bulletproof or it could be thrown away in favor of a system-provided
event facility. We provided our own log as a basic example of how to manage an
operational characteristic through a directory service.

Availability

The server automatically restarts after a restart interval expires, as shown in
Figure 26.7. The interval, set by an administrator, is obtained from a directory
attribute. Its purpose is to give a disruptive stimulus time to dissipate or to separate
resource-intensive restart attempts. The administrator may choose a delay of zero.
This restart attempt could increase service availability. A decision to intelligently call
the server DOA and quit the restarts after several attempts is missing. We simply quit
after five tries, even if they are days apart.

Favorably, multiple service points may be published. This enables static partitioning
of the client load across server machines, especially where the client service binding

```
22:14:35 TID 1576: halting
22:14:35 Worker 1576 ended
22:14:35 TID 1940: halting
22:14:35 Worker 1940 ended
22:14:35 TID 580: halting
22:14:35 Worker 580 ended
22:14:35 TID 2056: halting
22:14:35 Worker 2056 ended
22:14:35 Boss ended
22:14:35 Server ended
Launcher watching quoter...
22:14:37 'q' ends server
22:14:37 Boss running
22:14:37 Worker 1576 started
22:14:37 Worker 1468 started
22:14:37 Worker 1420 started
22:14:37 Worker 1844 started
```

Figure Z.6  Automatic restart
names are deployed centrally. An example is a client binary residing on a file share that is accessed through a centrally administered desktop shortcut. A browser-based scenario is a client under complete central deployment control.

Availability is also a function of serviceability.

**Serviceability**

Our quotation clients connect through a level of indirection within a service binding name attribute. This is published within a service control point in the directory. Each client interaction is relatively short. The connection to the service may be dropped after each request. A central administrator may install a new level of the quotation server and then redirect the service control point to reference it. The operation may take an instant, with no interruption of client service. Assume that the new level of server is tested off-line beforehand. If problems are detected on-line the procedure is reversible. Our omission is that we didn’t record the service level of the server in the directory. Please do this in enterprise applications, and provide an administration tool that knows how to manipulate it.

How do we roll out new client versions to gross numbers of users? A base method is to publish the clients as file shares or as three-tier Web-based applications. The client executable may be upgraded. Carry this out by locking it and swapping in a replacement. Change a Web page if it is browser-based. The client code level should be maintained in a directory object. We sidestepped this issue in the quotation example, but it’s a simple next step to carry this out.

**Scalability**

The application uses a pool of reusable stateless threads that lends itself to a larger number of simultaneous requests than a model of creating and destroying a thread per request. The application does not dynamically adjust the pool size based on past history and current workload—a limitation to scalability. This is offset by the promise of spreading servers across many machines and partitioning the set of clients such that they query the directory for the various service names. This is a form of load partitioning, not dynamic load balancing.

**Security**

Each client must find a server through a service control point attribute in the directory. That attribute may be protected by an ACL to limit who can use the application. LDAP accepts authorization credentials during the LDAP BIND API and its friends. Administrator access is controlled this way.
Summary

We’ve seen that a simple two-tier or three-tier application may be directory-enabled with a minimum effort once the problem is understood. We used the principles discussed in Chapter 13 to enable a simple, older application for use with a directory without accessing its source code. We’re able to demonstrate central administration of our server by storing operational parameters in a container in a directory. Clients locate the directory automatically, due to the default-binding behavior of LDAP on Windows 2000 Active Directory.

Our demo application stands up fairly well against the enterprise principles, considering the minimal directory enabling that we carried out. We know what we have to do to improve our application where it is deficient. The source is on the CD for those who want to carry this out.
There is no such thing as a free lunch.
—ANONYMOUS

Single sign-on means different things to different people. For this case study, single sign-on means only having to provide a name and password once. That means you sit down to your workstation and log on, but never have to provide more credentials again, regardless of where on the Internet or intranet you might electronically roam. This is not to say that you have free rein over the financial systems of the world, only that applications can recognize who you really are and act accordingly. After all, how many times should you need to prove your identity? This chapter presents the difficulties of implementing a single sign-on solution for an enterprise.

Internet and Intranet Users

The first issue that must be considered in embarking on the quest for single sign-on is whose sign-ons do you actually want to reduce. The first thing to do is to separate Internet from intranet users. The two categories of users have different requirements, concerns, and willingness to tolerate problems. (See Table 27.1.)

The comparisons in Table 27.1 demonstrate that single sign-on is a worthy goal for both Internet and intranet users.

A related concern, which must be included in the evaluation of any single sign-on proposal, is how much proof of identity is really needed. While a bank might require strong proof of identity for people moving large sums of money, a club newsletter much less proof. One common obstacle to deploying single sign-on solutions is the tendency to require it to solve existing security problems.
<table>
<thead>
<tr>
<th></th>
<th>Internet Users</th>
<th>Intranet Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consistency of systems</td>
<td>The only point of consistency for Internet users is that most have a version of</td>
<td>While no enterprise has uniformity across all of its end-user systems, most can</td>
</tr>
<tr>
<td></td>
<td>Windows. Any solution that needs consistency in the user's system will not succeed</td>
<td>create a general level of consistency. Even a small amount of consistency can assist</td>
</tr>
<tr>
<td></td>
<td>on the Internet.</td>
<td>in deploying client software.</td>
</tr>
<tr>
<td>Willingness to tolerate</td>
<td>“Two clicks and you’re out.” Internet users will go elsewhere if your site is</td>
<td>They are employees. While it might be painful to have multiple user IDs and</td>
</tr>
<tr>
<td>multiple sign-ons</td>
<td>hard for them to use. In addition, most people don’t like entering passwords.</td>
<td>passwords, it isn’t something most people quit over.</td>
</tr>
<tr>
<td>Tolerance for security</td>
<td>Customers and Web site visitors do not like having to sign on. However, they will</td>
<td>Employees normally understand that there are business reasons behind most security</td>
</tr>
<tr>
<td>measures</td>
<td>do so for compelling content or to reduce financial risks. In general, the user's</td>
<td>policies. (Well, at least behind good security policies.) Knowing that items are</td>
</tr>
<tr>
<td></td>
<td>willingness to put up with overzealous security measures is very low.</td>
<td>critical to the success or even survival of the company (that provides their</td>
</tr>
<tr>
<td></td>
<td></td>
<td>paycheck) increases their willingness to tolerate inconvenience.</td>
</tr>
<tr>
<td>Number of user IDs and</td>
<td>Once you could surf for weeks, or at least hours, without having to enter a</td>
<td>Classic systems usage has often led to people needing multiple user IDs. The</td>
</tr>
<tr>
<td>passwords</td>
<td>password. Now many sites collect and use demographic data about their visitors,</td>
<td>proliferation of intranet Web sites is starting to increase the number of user</td>
</tr>
<tr>
<td></td>
<td>which has led to an explosion in the number of sites requiring sign-ons. (Luckily</td>
<td>IDs required.</td>
</tr>
<tr>
<td></td>
<td>most sites use cookies, so you only have to reenter the password occasionally.)</td>
<td></td>
</tr>
<tr>
<td>Ability to make the problem</td>
<td>A single enterprise has limited ability to “solve” an Internet user's single</td>
<td>Tractable, in that the majority of the systems will be under the enterprise's</td>
</tr>
<tr>
<td>significantly better</td>
<td>sign-on problem. This is because the Internet user will have user IDs from many</td>
<td>control. In addition, given a smaller number of IDs, any improvement has a bigger</td>
</tr>
<tr>
<td></td>
<td>companies.</td>
<td>effect.</td>
</tr>
<tr>
<td>Ability to remove multiple</td>
<td>Internet users use a limited number of protocols when interacting with an</td>
<td>Companies have a multitude of systems and types of systems. This makes it hard,</td>
</tr>
<tr>
<td>sign-on with a single solution</td>
<td>enterprise. This makes it easy to solve the enterprise piece of single sign-on</td>
<td>maybe even impossible, to find solutions that cover the majority of systems.</td>
</tr>
<tr>
<td></td>
<td>with a single solution.</td>
<td></td>
</tr>
</tbody>
</table>
The Single Sign-On Solution

The best single sign-on solution for an application is not to require a single sign-on. Many applications would be improved if they replaced their private user sign-ons with existing operating system logons. Operating system logons use authentication methods that build on existing credentials. One example is Kerberos. Kerberos allows multiple applications to do secure authentication without requiring the user to enter multiple passwords. Appendix E provides a description of two APIs that allow application to leverage Kerberos authentication. Sample code is provided in the Appendix E section of the CD. Another method is to use underlying communication that already supports single sign-on, such as DCOM, OSF DCE RPC, and CORBA with IIOP. New applications should be designed using these technologies. That would at least prevent the single sign-on problem from getting worse. However, most of these technologies do not extend well to the Internet environment. They require users to be defined at a security system before communications can occur. The ad hoc nature of many Internet communities makes this a problematic issue.

Today, Web sites implement single sign-on by having the client system save and return a cookie that can be used to identify the user. This allows a user to register once and never to have to enter their password again. “Never” is an overstatement, but only having to enter a password after you have replaced your system is nicer than entering it even once a day. Cookies do not provide a real strong guarantee of identity, especially in households where multiple people use a single computer, and they must be constructed in such a way as to prevent attackers from manufacturing or stealing the values that allow them to pretend to be other users.

Public key certificates have emerged as the leading cross-company authentication method on the Internet. An individual can use a single certificate to enter multiple Web sites, potentially controlled by different organizations. Also, most public key systems allow multiple certificates to be stored and unlocked using a single PIN or password. A major advantage of public key systems is that users can carry their identity with them to new servers. The advantages of public key certificate are leading many enterprises to implement internal public key infrastructures.¹

If an application cannot use common, preexisting credentials, then password caching can be used to reenter the user’s password. This requires that some form of password synchronization be used as well. Once you have password caching and synchronization, the next step is to expand to nonuniform user name transformations. These technologies work like password caching except that they send a different user ID and password. Mapping can also be done on the server. For example, Windows 2000

¹. Lotus Domino uses a public key infrastructure internally, which allows Notes users to access Domino servers easily without having to reenter passwords.
can accept Kerberos tickets from other realms and will map them to Windows users. Many other systems also support name transformations. OS/390 can map many types of foreign user IDs to RACF user IDs.

**The Directory Solution**

Directories can help improve the single sign-on situation by

- providing a location for public key certificates and related revocation lists
- providing a location for storing mappings and credentials
- acting as an authentication service

Each of these cases requires a different set of schema classes and attributes. Storing public key certificates requires objects that enable the identification of the people’s public keys. The desire for revocation creates the need for objects that contain the identities of revoked certificates.

Using a directory to store identity and credential mappings requires the ability to associate multiple, system-specific identities to a single user and store the associated authentication data. The IBM common schema has a number of object classes to facilitate this type of directory usage, including the eUser and eAccount classes.

The directory service can also be used as the authentication service. This requires the directory to have the same concept of user as the system requesting the authentication service. Satisfying this requirement causes directories to need multiple password attributes (one for each system that requires a different one-way hash) and to have attributes that represent system specific types of data (like Unix uids, or Windows NT profile paths).

The size of the directory infrastructure required varies with how the directory is used. If the directory is the key authentication service, it will need to be highly scalable. If it is involved in some ID and credential mapping, it will not need to be as large. In addition, if it is used primarily for public key certificates and revocation lists, it can be comparatively small.

A starting point for achieving single sign-on is for application developers to integrate support for Kerberos and public key support instead of application-proprietary methods. Applications should also move from using older authentication services like Windows NTLM. In the meantime, systems need to enable more integrated security management for keeping multiple IDs in sync and translating between them. Older legacy protocols will be around for a while longer. (And unfortunately, new legacy protocols are invented every day.) The final ingredient is increasing the integration between security services and directories.
Summary

Directories are currently used in single sign-on solutions and will be integral to future single sign-on solutions. Using an enterprise directory to store and manage user definitions across multiple systems decreases administrative overhead while it improves user satisfaction. The tighter integration of common security solutions into applications will improve application security by reducing the chances for security coding mistakes.
Chapter 28

**Configuration Management of a Web Farm**

*Problems cannot be solved at the same level of awareness that created them.*

—ALBERT EINSTEIN

Many Web sites are actually Web farms. A *Web farm* spreads the load across many servers, while maintaining the illusion for visitors that they have come to one spot. Some Web farms are actually geographically dispersed collections of server machines. Managing a Web farm is thus managing a distributed networked resource. A directory is a distributed networked data store, so a directory is a natural solution to the enterprise problem of managing a Web farm.

### The Web Farm Problem

A real Internet Web farm is the msnbc.com news site. At one point msnbc.com consisted of 35 servers. Four servers were primarily DBMS backend data servers. Twelve servers were shared with other sites. Some servers were reserve servers brought on line for peak news events where the normal load of 20,000 concurrent visitors would balloon to 100,000 concurrent visitors. Sites are dynamic—MSNBC is now probably a different mix of numbers of hits and servers as technology explodes and visits grow.

A cyber-soiree using the DNS utility, *nslookup*, yielded the information in Figure 28.1. DNS address records for six IP addresses resolve to msnbc.com. The reader may try this nslookup technique on a number of popular Internet sites to get a feeling for how many of them are Web farms. This isn’t a definite check, because even when a domain name resolves to one IP address the site may have a Web farm behind a single IP address going into a load-balancing server or cluster.

---

Some Internet service providers operate a Web farm configured as many IP addresses. Each server in the ISP farm may answer to all of these addresses, or the address space may be partitioned among the servers. Figure 28.2 shows an nslookup interrogation of Yahoo’s popular free ISP, Geocities. Another type of site may have many domain names but only a single IP address.

We shall explore configuration management of a Web farm by proposing a hypothetical, entry-level Web farm.

**Web Farm Design**

Our example Web farm consists of two or more Web servers, a DNS, and a backend data store residing in one or more servers. We shall position our installation as a low-end site with room to grow. Allowing for growth is important because it is difficult to predict when and how much workload growth will occur.

The Web farm appears to the user as one URL. Each browser request that resolves the host name of the URL may receive a different IP address than another request. The DNS is used as a simple IP address “sprayer.” It is set up to resolve each sequential host name request for the Web farm URL to the next server, round-robin fashion.

Our Web farm is not restricted to serving up static HTML pages. It may run some intranet business process applications or Internet e-Business applications. When we talk about applications on a Web server, we need to address the stateless nature of an HTTP request. Stateless servers are at once a scalable blessing and a programmer’s curse. An intranet or Internet application session with a user usually needs to maintain state-per-user across several HTTP request cycles. This state is called user session data. This problem is usually handled in single-server sites by either keeping session data in each client browser or keeping it in a server-side database.
Imposing a Web farm on a site adds the general possibility of a different Web server handling each request of a client’s series of requests during a session. This adds more difficulty to the session data problem than if a server-side state maintenance scheme is chosen. Some possible design alternatives to address this state problem are as follows.2

1. Design the application to be stateless. This is not possible with some applications. When it is possible the application may scale well. Each request is treated as if it came from a new and distinct user. A search application is an example. We reject this approach for our Web farm application because we stipulate that we cannot constrain all of them to this model.

2. Maintain state in a DBMS server (or a DBMS cluster for fail-safety) within the Web farm. This adds to site server and data requirements. This is a good approach for applications that have large runs of state data. Our Web farm applications have simple state requirements—just a few variables are needed. We won’t adopt this option either.

3. Use a load-balancing technique that is aware of user sessions. Once an IP address is assigned to a request, that IP address always handles subsequent requests for the rest of that session. The server at that IP address must maintain session data for the request. We’re using round-robin DNS for our load-balancing server. We could assign separate, unique names to each member Web server in the farm. When the DNS assigns a server, that server would handle the rest of the session, provided that all URLs in the application’s pages are relative URLs so that no more DNS resolution is needed. Otherwise the session would be lost if another server were assigned by the DNS in mid-session.

---

4. Maintain state in each client’s browser. This is a common technique on single-server applications. Fortunately, it works on Web farms also. The only downside is a limitation of the size (about 4KB) and nature (flat data) of the data that may be cached. Flattened object streams are usually not appropriate here. Our Web farm uses a few numbers and strings for its application state variables. This option looks like a fit for us.

There are variations of the fourth option, such as using hidden form fields to pass the state to the browser. The server sets the state into hidden fields of a form it is about to serve to the browser. State comes back to the server when the form is submitted. We like to use HTTP cookies instead because they are less cumbersome. (See Figure 28.3.) Cookies contain name-value pairs that have a life span set by the server application. The server sets the initial value into the cookie as in the hidden field approach. The cookie travels back and forth with each HTTP request and response. (See Figure 28.4.) The cookie disappears after a server-determined expiration time has elapsed. No HTML forms are required, but they are probably used in part of the application. The downside of cookies for an Internet application is that the user may turn off cookie support in the browser. Of course the application can inform the user why it cannot proceed and explain how to turn cookies on.

Let us recap how we maintain session data. Our Web farm applications do not lend themselves to being stateless. They do not require huge runs of session data. We don’t want to impose a session-aware load-balancing technique on top of our DNS round-robin host name-to-IP resolution. We choose the fourth option as our method of saving session data. We drop a cookie to the requesting browser at the end of each request cycle within a session. The cookie, a name-value string, is automatically sent back to the server with each succeeding request. It doesn’t matter which server gets the request and its cookie, because state is set from the cookie.

We have overcome the session problem exacerbated by expanding a single Web server to a farm of Web servers. We have conceptual lift-off of our Web site without a directory service but we still must solve the problem of managing it before we can really start.

**HTTP Cookies**

- **Cookie**: small piece of data set by server into browser, keyed per URL.
  - **Set-Cookie**: name=aValue; expires=date; path=aPath; domain=aDNSName
- **Browser** gets cookie back in HTTP header
  - **Cookie**: name1=somestring; name2=otherstr
- **Used for session data**
- **Values determined by server application**
- **Exist until expiration (or user clears)**

**Figure AB.3**  Browser cookies
Each server in the Web farm has identical content. The content is updated at frequent intervals. Some news sites update their content as often as 24 times per day. Each update must proceed without user perception no matter what the update frequency.

**Adding Servers**

Some Web farms can add more servers when infrequent peaks of activity occur. For instance, a weather site may have a flat workload most of the year until a hurricane season spawns one or two killer storms. Everybody on the Eastern and Gulf coasts of the United States suddenly becomes interested in the weather. People in potential target areas yearn for the slightest hints about where the hurricane may strike. The weather site’s Web farm may suffer two orders of magnitude more hits with very little forewarning—this is the nature of hurricanes. The weather site Web farm staff responds by adding servers. This is easy to say, but how is it carried out?

The staff needs to be able to replicate current content onto new servers and slip them into the system without disturbing operation. These servers must immediately pull their load with no adverse side effects.

When the crisis has passed, these reserve servers may be retracted from the Web farm. They may be rental machines or they may normally fulfill less-important server tasks, such as rotating advertising banners.

How does the staff know when to add servers and when they are no longer needed? Each member Web server logs activity. This consists of requests and responses with timestamps as well as errors. This data must be regularly analyzed. The Web farm
staff must be notified somehow when performance moves above or below separate, established thresholds. Some distance to provide hysteresis to avoid “chatter” must separate these thresholds. Usually a log analysis agent daemon pages the Webmaster when one of these bogies is reached.

**Managing the Web Farm**

Managing this aggregate we call a Web farm involves these tasks and operational characteristics.

- Centralized control of the constituent servers
- Transparent replication of content updates
- Web service advertising: load-balance-mapping of DNS to correct IP addresses
- Performance monitoring of individual servers
- Performance monitoring of the entire farm
- Crisis notifications to human operators
- Hot-plugging servers and removing servers
- Backup and restoration of backing store without service interruption
- The ability to piecewise migrate the individual boxes to other platforms if needed

Let us keep these goals in mind as we design the Web farm. Then we'll revisit them, using directory service glue to meet these goals.

**Implementing the Web Farm**

We have a small budget. Maybe after our site proves to be indispensable, we’ll get a larger budget, but for now we want an economical solution without sacrificing reliability or price performance.

Intel-based Linux is becoming popular in Internet and intranet environments where low-end servers are used. We choose Linux on Intel servers as our operating system. The Apache Web server is ubiquitous across platforms and it comes with most Linux distributions, so we’ll use Apache for our Web servers. It is configured from a script that resembles a Windows INI file. We’re starting out with two Web servers, but our design must allow for a good measure of growth. We shall lay out our farm such that more servers may be added. We could port to another flavor of Unix on other hardware later, as long as we keep platform-specific code to a minimum.

We need to manage the aggregate service as a whole, so we’ll use an LDAP directory server to glue our conglomeration together. Our example uses a port of University of Michigan slapd and slurpd provided by the OpenLDAP organization. (See Figure 28.5.)
Slapd will be our LDAP server and slurpd is the replication agent. We’ll use more than one copy of our management directory to eliminate a possible single failure point. Slurpd will maintain loose consistency between the replicas.

At first glance slapd seems intimidating. It is shipped as source code, unless you get it bundled within a Linux distribution. That is the price of a freebie, but it is also a benefit. One builds it for a given platform. It turns out that this is not too difficult on a full Linux system. OpenLDAP has a configuration script that sets options in environment variables. It has sane defaults that will work for us. It probes the build environment and assumes the target platform is that machine. We run the configuration script. Next we make dependencies followed by a make. When it works we run “make tests.” Extensive tests involving populating the directory, adding, deleting, searching, and replicating are carried out. Logs are generated. It works! The most difficult part will be configuring the directory service and adopting a schema. We’ll come back to how we’ll use the directory after we decide the function units of the Web farm. We’ll decide upon a simple schema at that time.

We’ll need to dispatch client browser requests between our Web servers. We’ll set up our DNS to resolve successive host name resolution requests, round-robin fashion, to each participating Web server. This coarse load assignment scheme is non-adaptive but it works well for low-end Web farms. The DNS daemon used throughout most of the Internet is from Berkeley and is called BIND. It is in the public domain. We’ll ensure that we’re using BIND 4.9.3 or later. Most current Linux distributions use either BIND 4.9.7 or one of the BIND 8 releases, so we’re in good shape for assigning IP addresses round-robin to the servers. All that BIND needs is an IP “A” record for each IP address to be assigned to our Web farm domain name. Each Web server will run a copy of BIND with an identical forward zone file, as shown in Figure 28.6. The reverse zone will be different for each Web server because each has a separate IP address.

Directory services are commonly used to publish and locate services for clients. DNS is actually a directory service, albeit an LDAP-ignorant service. We’re actually using

---

**OpenLDAP**

- Build binary slapd or install with rpm
- Edit /etc/openldap/slapd.conf: e.g.
  - suffix o=Web-Farm, c=us
  - rootdn cn=root, o=Web-Farm, c=us
- Use an LDIF file to populate the directory
  - ldapadd -D 'cn=root, o=Web-Farm, c=us' -f load.ldif

---

DNS to publish and locate our Web servers here. In turn, we’ll publish the DNS services in our slapd LDAP directory for use by administrators. Keep this in mind when we design the schema.

The directory service is the locator service for everything except for the directory service. This is not a problem. We use our other directory service—the DNS—to locate our directory replicas. We create an alias for www.thefarm.com named ldap.thefarm.com. Our initial design keeps a replica on each Web server. Simply changing the DNS reference to it will relocate the directory service.

We shall be implementing dynamic content emitted from three-tier applications. These will need a backend database. We may as well stick to Linux for now. There are a growing number of DBMSs for Linux. For instance, IBM ships DB2 with some of the commercial Linux distributions. For now, we shall use PostgreSQL, which ships with many Linux distributions and supports atomic database updates protected by transactions. We may move to another DBMS later, but the move will have to encompass all servers or none because we are not using an external transaction monitor, such as CICS or Tuxedo, that would allow multiple types of DBMSs to participate in a transaction. We must remember to publish the administrative and service location binding information for whatever DBMS we use in our directory. Our middle-tier applications will use the directory to locate the DBMS. For now, we’ll use just one DBMS instance. This gives us a potential single point of failure. We make this compromise because we don’t want to replicate DBMS updates tightly. We soften the problem a bit by allowing for redirection to a backup DBMS server through the directory service binding information. This begs the question of how the stand-in gets its data. A periodic backup operation will be used to export the data store to the backup server.

Figure AB.6 DNS zone file configured for round-robin address resolution

```plaintext
@   IN   SOA  www.thefarm.com. (  
1999120502  ; Serial
10800     ; Refresh seconds
7200      ; Retry seconds
10800     ; Expiry seconds
86400     ; Minimum TTL seconds
)
; Name server record
PTR localhost.
; Address records for member servers in our farm
; Records for www.thefarm.com will be used round-robin IP resolution.
www.thefarm.com  A  9.37.208.194
www.thefarm.com  A  9.37.208.212
...
```

DNS zone file configured for round-robin address resolution
This isn’t a 24 x 7 stance but it does stave off complete disaster if the DBMS machine fails. A watchdog script would periodically check the DBMS logs and PID, locating them through the directory, of course. It would page an administrator if it detected a DBMS malfunction and automatically switch to the backup database by changing an attribute in the service binding directory information for the DBMS.

Our middle-tier business logic will reside in CGI applications. Such applications may be implemented in C or Perl. Perl is quite popular for Unix Web applications. The performance penalty of many short processes per CGI session is often overcome by using a form of CGI that uses reusable processes. Up to two orders of magnitude of performance improvement over straight CGI may be realized. Perl may be used with a library called Mod Perl with the Apache Web server to create reusable Perl CGI program processes. Each is compiled on first use and pooled for reuse by an Apache filter module.

One option is to use Active Server Pages (ASP) in the middle-tier logic. ASP, a feature of Microsoft’s NT and Windows 2000 Internet Information Servers, is a form of server-parsed scripting featuring HTML marked up with scripting tags. The server executes the script as it emits the HTML stream to the client. A server-side object uses cookies to maintain session data in the client browser. The session object is reinitialized from the cookie at each interaction, even if each interaction occurs in a different Web server within the farm.

We make no constraint on the type of middle-tier program technology we include. It could use traditional CGI, fast persistent CGI, Active Server Pages, Mod Perl, Java servlets, or Java server pages. Our stipulation is that middle-tier logic must always consult the directory service to locate or bind to services such as the DBMS. No application caching of these locations is allowed. Straight CGI cannot easily do this anyway. Consulting the directory gives the Web farm fail-safe resiliency and enables an administrator or watchdog script to redirect service. There should be no performance hot spot because good directories cache requests unless the target attribute changes.

Much of our prototype directory administration will be carried out through scripts that we construct. Unix-Linux scripting offers many options, including choosing to create X11 GUI scripts. We will design command-line tools but hold out the option to grow these into pretty GUI tools.

Management Objectives

Design is an iterative procedure. We stated our management objectives, then laid out some high-level architecture. Now it’s time to march down the list of management objectives and fill in the blank spots. We’ll use scripting to carry out some tasks in concert with indirection through the directory. Afterward, we’ll choose a directory schema.
Centralized Control of the Constituent Servers

We have two Web servers, two primary DNS servers, and one backend data store DBMS server. A reserve DBMS server is periodically updated. We shouldn't forget our directory service servers. We'll put replicas of our directory on separate servers. Our proof-of-concept configuration will be two machines. We can grow these assignments later without service interruption, because we use indirection through the directory and the DNS. See Figure 24.7 for the big picture.

Apache Web servers are administered through a configuration file for each server. These files would be shared through NFS, with careful access control. They would be advertised through a service administration point in the directory. A central administrator may alter configuration parameters in each server individually. This administration tool could also be made into an alternative Web application that may be run anywhere using an SSL connection.

- Client uses a machine IP assigned through round-robin DNS resolution.
- Apache-based application finds DBMS through slapd request at each request.
- DBMS reassigned at any time by an LDAP update to slapd service-binding entry.

![Diagram of machines and services](image-url)
Transparent Replication of Content Updates

The possible Web server content consists of static text files, textual scripts or active server pages, Java server pages, binary CGI applications, and so on. These files will be shared through service administration points in the directory service. Updates may be carried out through these shares. We must remember to keep all content identical across Web servers because a Web farm appears as one server. We must not interfere with executing user sessions. One tactic is to pause the Web servers, update the content, and then resume the servers. Content updates are usually incremental changes, so an update happens in an instant. A user may notice a response-time peak but no real interruption of service.

The backend data store may also need to be updated at times. This is carried out like any update caused by an application. DBMS locking and transaction protection makes the update safe and smooth.

Web Service Advertising: Mapping of DNS to Correct IP Addresses

We’re using a nonadaptive dispatching technique that is almost free. We advertise our Web farm domain name in a DNS and let a feature of BIND 4.93 or greater dole out IP addresses round-robin fashion. This simple tactic should work well provided we anticipate running short-duration applications that are not resource-intensive. We do not stipulate affinity to a processor or machine during a user session. This means that successive HTTP requests of a session could be handled on different machines.

If for some reason an application did want to keep all of its requests on the machine on which it was initially dispatched, it could use only relative URLs within its forms and pages. The server would not make a subsequent name resolution request to DNS, so the same IP address would continue to be used throughout the session. This relative URL restriction seems error-prone, but we don’t preclude it.

Performance Monitoring

The Apache Web server keeps configurable, extensive logs, which we can leverage. The PostgreSQL server also does logging.

The logs on all machines will be on NFS shares for remote access—secured of course. We will find all log shares through service points in the directory.
Crisis Notifications

A daemon Perl script will awaken periodically to reference the directory to find the Apache and Postgres logs. It may analyze the logs and conditionally trigger an RF pager message to an administrator. The location of the logs and the trigger thresholds are kept in the directory.

Hot-Plugging and Removing Servers

Personnel may install a server machine, configure software for it, and test it offline. When it is ready, the server with all of its services would be brought on line by updating attributes in the directory. Our rule is that everything that needs a service finds that service through the directory—nobody caches this stuff except the directory service.

For example, a new Web server machine would have its IP address added to a service location point in the directory. The DNS would need an address record for it also. The server would also have an entry added to a service administration point. The NFS shares for its logs would be referenced in a multivalued attribute.

A server machine could be backed out of the Web farm by simply changing a few attributes in the directory and perhaps the DNS. Here, the operators must watch logs or other monitors to ensure that all users are drained from the system before they physically disconnect it.

Backup and Restoration of Backing Store Without Service Interruption

We choose to have a reserve server running PostgresSQL. Our approach is to export the primary data store to the reserve server, while continuing to run the primary server. We restore the backend data store by making the reserve server the primary server. Since the reserve server is only updated periodically, this could be a problem for update-intensive applications. This is the largest hole in our Web farm so far. The fix would be to use a DBMS replication provided by a commercial database. There are free trial versions of these for Linux. We choose to keep our current server design so that we can prototype our Web farm from a single Linux distribution installation.

---

4. DBMS replication is provided by DB2 UDB for Linux, available for a 60-day trial from IBM.
Migrating Individual Boxes

We’ve already satisfied ourselves that we can add and remove boxes or reassign software processes without service interruption. Our ability to migrate to other hardware or software platforms is limited only by how our services map elsewhere. Services on both sides of a migration must continue to cooperate if we are to consider 24 × 7 (maybe 23.9 × 7) operation.

Moving the DNS isn’t likely to be a problem. It is the glue that binds the Internet without particular regard for hardware or software platforms. DNS is truly equal opportunity. We may move our DNS to another operating system, for instance, and continue to operate. The DNS only must continue to be authoritative for resolving the zone for domain www.thefarm.com. This means it has the final say on what IP address is resolved for the zone.

The slapd and slurpd services from OpenLDAP aren’t an issue in migration because they can operate on many other platforms. (Alternatively, another directory service may be substituted.) IBM’s Secureway Directory Server, Netscape’s Directory Server, Novell’s NDS, or Microsoft’s Active Directory are LDAP directories. Our Web farm could use one of them, provided that our schema is exported to a suitable partition in one of those directories. Migrating the directory service to another directory implementation while keeping the farm alive may require some planning. LDAP and the LDIF export format are enablers. Our schema should import to another directory without incident unless it is to be merged with a larger schema.

Schema

We know how our hardware and software boxes fit together. We recognize that we’ll have to write scripts to leverage the directory in our installation. We’ve stipulated that all middle-tier applications must use fresh directory references to find services. Now we can design our schema keeping in mind how we want to use the directory service. This Web farm is a proof-of-concept effort. We’ll use the schema shown Figure 28.8. A larger solution would extend or subclass an existing schema.

The Directory Infrastructure

By following the instructions when installing slapd and slurpd from OpenLDAP, we can build a workable directory, test it, and install it. Unix man pages are also installed. The slapd daemon should be started by a line in rc.local such as slapd &.
The program identifier (PID) of the slapd server resides in /usr/var/slapd.pid when the service is running.

LDAP client utilities are included with the OpenLDAP distribution.

- ldapadd
- ldapdelete
- ldapmodify
- ldapdpasswd
- ldapdsearch

Each has a man page and emits help when passed the switch --help. These utilities may be called by scripts. One may create an LDIF file and use ldapadd to populate a directory. The ldapsearch utility is the most-used utility. Our scripts may use it to access service and administration locator attributes. A debugging trace is emitted to stdout if a debugging parameter (-d) is passed on the command line.

Directory Configuration

Our directory configuration is shown in Figure 28.9. We shall configure slurpd to replicate between our two slapd directories. When a replication wait interval expires, slurpd locks a log file of updates and forks a child replication process for each configured slapd server. Each child runs through the log file, updating entries to its assigned slapd server. Each child process coordinates through timestamps such that the last update wins across the replicas. This all happens while users are active on the system. Administrative updates that occur during the cycle are saved for the next cycle. We didn’t need to do anything except configure slapd and slurpd for all of this to be carried out. This is slick stuff!

Figure AB.8  Schema for Web-farm.localdomain directory service

- We re-use the University of Michigan sample schema
  - This is not recommended for “real” applications
  - Path of least resistance for a sample demonstration
- We subvert the “service” object class
  - Attribute “provider” advertises service IP address
  - Attribute “serviceArea” advertises service IP port
  - Syntax is case-independent-string (CIS) for each
- Distinguished name of service connection:
  - dn=cn=dbms, cn=root, o=Web-Farm, c=us

The program identifier (PID) of the slapd server resides in /usr/var/slapd.pid when the service is running.

LDAP client utilities are included with the OpenLDAP distribution.

- ldapadd
- ldapdelete
- ldapmodify
- ldapdpasswd
- ldapdsearch

Each has a man page and emits help when passed the switch --help. These utilities may be called by scripts. One may create an LDIF file and use ldapadd to populate a directory. The ldapsearch utility is the most-used utility. Our scripts may use it to access service and administration locator attributes. A debugging trace is emitted to stdout if a debugging parameter (-d) is passed on the command line.

Directory Configuration

Our directory configuration is shown in Figure 28.9. We shall configure slurpd to replicate between our two slapd directories. When a replication wait interval expires, slurpd locks a log file of updates and forks a child replication process for each configured slapd server. Each child runs through the log file, updating entries to its assigned slapd server. Each child process coordinates through timestamps such that the last update wins across the replicas. This all happens while users are active on the system. Administrative updates that occur during the cycle are saved for the next cycle. We didn’t need to do anything except configure slapd and slurpd for all of this to be carried out. This is slick stuff!

Figure AB.8  Schema for Web-farm.localdomain directory service

- We re-use the University of Michigan sample schema
  - This is not recommended for “real” applications
  - Path of least resistance for a sample demonstration
- We subvert the “service” object class
  - Attribute “provider” advertises service IP address
  - Attribute “serviceArea” advertises service IP port
  - Syntax is case-independent-string (CIS) for each
- Distinguished name of service connection:
  - dn=cn=dbms, cn=root, o=Web-Farm, c=us

The program identifier (PID) of the slapd server resides in /usr/var/slapd.pid when the service is running.

LDAP client utilities are included with the OpenLDAP distribution.

- ldapadd
- ldapdelete
- ldapmodify
- ldapdpasswd
- ldapdsearch

Each has a man page and emits help when passed the switch --help. These utilities may be called by scripts. One may create an LDIF file and use ldapadd to populate a directory. The ldapsearch utility is the most-used utility. Our scripts may use it to access service and administration locator attributes. A debugging trace is emitted to stdout if a debugging parameter (-d) is passed on the command line.

Directory Configuration

Our directory configuration is shown in Figure 28.9. We shall configure slurpd to replicate between our two slapd directories. When a replication wait interval expires, slurpd locks a log file of updates and forks a child replication process for each configured slapd server. Each child runs through the log file, updating entries to its assigned slapd server. Each child process coordinates through timestamps such that the last update wins across the replicas. This all happens while users are active on the system. Administrative updates that occur during the cycle are saved for the next cycle. We didn’t need to do anything except configure slapd and slurpd for all of this to be carried out. This is slick stuff!
Critique of the Solution

Reliability

Linux has a reputation of being a reliable operating system. We added to our reliability by eliminating some possible single points of failure. We chose to have directory replicas and more than one Web server. We want to keep a failure from taking down the entire Web farm. We glue Web servers and backend data store together though the directory and DNS, and by making all applications find services through the directory at all times.

A reliability weakness lies in the backend data store. We don’t have on-line data replication in our design. We do have a reserve server that is updated at intervals, but some applications may operate on stale data if the reserve is quietly switched on line in place of the primary DBMS. A replicated DBMS should be used if our Web farm starts to grow.

Availability

We expect somewhat increased availability because of the elimination of some single points of failure and our selection of reliable software. Our DBMS design does not hinder availability because we believe we may quietly replace the primary DBMS with

---

**Figure A8.9** Directory configuration

1. Client updates an attribute.
2. slapd updates the attribute at target directory.
3. slapd also writes timestamped update into replication log.
4. slurpd wakes up, reads information from log.
5. slurpd carries out LDAP update of the attribute in other slapds.
6. slurpd removes log entry.
PART 4  •  Internet and Intranet Case Studies

A reserve DBMS. The choice of IP assignment through cheap round-robin DNS weak-ens availability. Clients of a failing Web server will encounter time-outs because the failing IP address will remain in the client DNS resolver cache. A hardware network IP dispatcher substituted for round-robin DNS would increase availability.

Serviceability

We attained centralized administration and the ability to switch components of the Web farm. This gives us very good serviceability for the prototype Web farm.

Scalability

Once we licked the problem of keeping user session data while rotating Web servers, we were halfway done with the scalability issue. Then we registered the servers in the DNS for load distribution and provided service administration points in the directory. This now allows us to arbitrarily add Web servers, which may be geographically dispersed.

More directory service replicas may be added and extended geographically as well. The efficiency of the middle-tier applications has an influence on scalability. We may decide to host a number of Mod Perl applications. Some literature indicates that Mod Perl scales quite well.

The backend DBMS may not be as easy to scale up, except by choosing higher capac-ity hardware and a more hard-hitting DBMS. Some planning would be needed if this switch were to be made with users on line. We believe our design, which locates the DBMS through a directory service point, would be a helpful factor.

Security

Security is not a particular problem in our Web farm. Any good directory server pro-vides state-of-the-art choices in security and OpenLDAP is no exception. We may choose Kerberos or another common security provider that can interoperate with other platforms, including Windows 2000, AS/400, or OS/390, if we choose to mix-and-match later. The Apache Web server provides per-user login when needed. It will participate in SSL connections with browsers that support it.

Limitations

Aside from a weak reliability point in the backend data store, our solution is limited by the administrator's ability to program control scripts to administer and control the
glue between the directory services and the servers. This could also be viewed as a strong point by those that want the utmost freedom to control the Web farm design.

Another limitation is our choice of using round-robin DNS IP address resolution as an inexpensive network dispatcher. If one of the Web farm servers goes down, a client may still possess the cached IP address of the failing member server. If the DNS is updated to remove the failing IP address, the client may still try to use it. A better solution is to use a real network dispatcher or a hardware solution.

**Summary**

Many Web sites are really collections of Web servers. Load-balancing and distribution techniques are used to allocate workload among the servers of a farm. This presents the potential for a user session to be spread across more than one server. The session data must be maintained such that this isn’t a problem.

Introducing multiple servers adds scalability and reliability, and potentially impedes administrative flexibility. We use scripts and a replicated directory service to locate servers for administration. The DNS, another type of directory, is used to locate the Web servers and provide workload rotation. We cannot achieve $24 \times 7$ operation, nor true load balancing, because round-robin DNS doesn’t remove a failing IP address. A better approach is to invest in a hardware network dispatcher.

We miss having a real-time DBMS replica. A commercial database, such as IBM DB2, would provide a replication option. We use a standby server instead. Our proof-of-concept Web farm measures up to the enterprise principles except for these weaknesses. Our Web farm provides a good entry platform that may be improved by tackling potential single points of failure incrementally.
his case study describes the business and technical topics involved in a metadirectory project. Metadirectory services are used to tie together existing directory systems within an enterprise. Often the impetus for implementing metadirectory services is the desire to transform a company into an e-Business. This chapter describes a case study that focuses on that portion of an e-Business transformation in which directory and metadirectory play a pivotal role.

No one wakes up one morning and decides to implement a metadirectory. The business and technical requirements for metadirectory are not easily identified. Recognition that a metadirectory is needed is usually an iterative and lengthy process.

The PMW Metadirectory Problem

Piscataway Machine Works (PMW) is a fictional large manufacturing company. It has several plant sites and about a dozen sales and distribution warehouses across the United States. PMW has made a substantial investment in information technology (IT) over the last 25 years for all aspects of its business, from accounts payable (AP) and general ledger (GL) to parts distribution and replenishment, computer-integrated manufacturing (CIM), payroll, and human resources (HR) management, e-mail, and business intelligence (BI). The computer hardware inventory includes mainframes, minicomputers from several vendors, LAN file and print servers of various kinds, and several hundred PCs, all networked together. In other words, it's a typical enterprise IT infrastructure.

The Move to E-Business

PMW is in the initial stage of transforming itself into an e-Business. (See Figure 29.1.) It doesn't sell books or CDs, and nearly all of its customers are other companies rather than consumers. But an economic analysis study has shown that it can reduce the cost of sales and product support by 30 percent by allowing its customers to order products, view service bulletins, and submit questions and suggestions via the Internet.
There is even an opportunity to develop a new retail channel for its consumer-oriented machine tools for woodworking, which are only available to resellers today.

The initial technical proposal is almost complete. It includes deploying a sophisticated Web server with ordering and payment services, an Internet white pages and e-mail feature to allow customers to communicate with PMW sales and support employees about products or their orders, and a directory system to store all information that is fairly static, including employee white pages data, customer information, product and parts catalogs, and so forth.

**Round 1— The Answer Is the Web. What’s the Question?**

Jose, the e-Business project manager, is excited about his design and about the opportunity to use Internet technology to improve PMW’s business. Once his design
is done, he schedules the first project review with his peer managers from IT and the key business area managers. The presentation of the business goals is well accepted, but there are several issues and concerns with the design.

1. Fran, the customer accounts department manager, explains that her group is the only department in PMW that can create a customer information record. These records are contained in a VSAM file on the mainframe and created using a CICS application. Over a hundred programs read this record, but her employees are the only ones allowed to create or modify the data. A request to create a new customer account record is submitted by sales clerks at the warehouse locations and it is created by her department. That’s the way it’s always worked. Fran takes a firm position: the new Web system is not authorized to create new customer records or change existing customer information data. It is free to create its own customer list, but that list is not official and will have no connection to the existing IT system.

2. The proposed e-mail system has a new Web-based address book. It lists the name and e-mail ID of employees that deal with customers. There is no automatic link between this list and the existing e-mail system directory used by PMW. Andre, the support manager for the existing e-mail system, is more cooperative than Fran and is willing to export the data from the current directory into a file so the Web group can import it into the Web-based address book.

3. Graham, the HR manager, mentions that PMW’s customers will be very unhappy if they send e-mail to an employee and get no response. When asked to explain further, he says that the internal e-mail directory is always at least a month out of date and the occasional importing of information from the internal e-mail directory into the Web address book will only make the problem worse. When an employee joins the company he or she is given an employee ID by the HR department. Then there is a long list of processes to follow, some of them computerized, some not. Employees must be added to the payroll system, the employee benefits systems, and be assigned office space, parking, and an e-mail account. It takes three to four weeks to update the e-mail system. When an employee leaves, the HR group reverses the process. This also takes three to four weeks. PMW employees have learned to adjust to the dead e-mail accounts of former employees, but customers may not be so understanding. (See Figure 29.2.)

After the meeting Jose is understandably distraught. Critical information needed for his e-Business application is either out of date or not available. He has been told that new customers, the holy grail of e-commerce, are not even allowed into the IT system! It’s time to rethink the e-Business design to address these issues.
Round 2—The Answer Is Directory Architecture. What's the Question?

Jose has spent the last couple of years working on Internet projects. He has a good understanding of the benefit of open systems protocols and centralized directory services offered by LDAP-compliant directories. The infrastructure of his e-Business solution already includes an LDAP directory server which will be used to store information that does not change frequently, such as the employee white pages information for the Internet address book application and customer information.

All the applications that are included in his e-Business site will access this information using the same protocol. In addition, the directory name space has been defined
to eliminate multiple occurrences of the “same” data by judicious use of referrals. Jose’s team has even defined a security model for the data contained in the directory and designed an administration graphical user interface (GUI) to make it easier to maintain. This is a directory service design to be proud of!

Jose reviews the directory design based on the feedback he received at the recent project review meeting and identifies two major issues. He updates his proposal to define the issues and his proposal to solve them.

**Information About Employees** Information about PMW employees is needed by many existing systems and by the new e-Business Web address book. Currently the same (or very similar) information is entered into multiple application-specific directories. There is no programmatic relationship to these multiple stores. Synchronization is done manually, is not done in a timely manner, and is error-prone.

- Technical solution—All information about PMW employees should be stored in an LDAP-compliant directory.
- Implementation proposal—the directory server that is part of the e-Business solution will become the enterprise directory for PMW. All applications in PMW that require data about an employee will access it using LDAP calls. Extensions to the schema and name space will be made as required to ensure that the directory server can store all required data elements. This proposal will eliminate duplicate entries, and Jose estimates that it will also reduce the elapsed time for updating his Web e-mail address book from three or four weeks to one day.

**Information About Customers** Customer information records are located on a backend legacy system. The new e-Business application has a business objective of reducing the cost of sales for existing customers and has the potential for creating new customers. Existing business policy prevents customer auto-registration via the Internet. This makes selling to new customers more difficult.

- Technical solution—Information about customers should be stored in an LDAP-compliant directory.
- Implementation proposal—the directory server that is part of the e-Business solution will become the enterprise directory for customer information for PMW. Applications will access this information using LDAP calls. The security model will be changed to allow new customers to auto-enroll at first purchase. Extensions to the schema and name space will be made as required to ensure that the directory server can store all appropriate data elements. This proposal will ensure there is a single source for all customer data and will facilitate selling to new customers over the Web.
Jose feels that this is an architecturally elegant solution and makes excellent use of Internet-based technology. He’s ready to reconvene the project review meeting and present his project modifications. (See Figure 29.3.)

At the second meeting Jose’s proposal to store all information about employees in an enterprise directory is not well accepted. Graham explains that some HR information about employees is highly sensitive and cannot be included in a shared directory system. After Jose explains that his comprehensive security model supports fine-grained access control, Graham replies that HR is legally responsible for protecting the confidentiality of employee information, not the e-Business group. HR will not agree to centralize employee data.

This proposed solution won’t work for the existing e-mail system either, according to Andre. The e-mail package that PMW uses for internal communications is from an independent software vendor and is based on proprietary interfaces. It doesn’t have an LDAP API for accessing user IDs and related data, and PMW cannot modify it to add the required interfaces. Jose’s proposal is not acceptable to the e-mail support team either.

Objections are also raised to the second design proposal for centralizing customer information. Mike, the IT manager, has compiled a list of all the production applica-

---

**Figure AC.3**  All relevant applications connect
tions that currently use the CICS-based customer information file. It turns out that Fran was correct—128 programs read the file every month. The AP, GL and other existing business applications have no requirement to add or update information, so Fran’s security policy isn’t a problem. On the other hand, the conversion of 128 programs from COBOL and CICS to LDAP API is a major problem.

Mike has estimated the conversion task will take nine person-months, plus three person-months for testing. He does not have anyone available to work on this for at least six months and isn’t sure he can even find the source code for a couple of the older batch programs. He concludes that moving PMW to an enterprise directory service based on open systems protocols is a good idea in theory but isn’t very practical.

Fran is still not supportive of Jose’s design. She explains that the problem is not VSAM or CICS or mainframes or even her ownership of the creation of customer information data. There is a fundamental business policy behind the process for accepting new customers. PMW’s customers purchase, on average, $600,000 worth of products per year. Most transactions are 60-days cash and include a line of credit. Standard & Poors must evaluate the credit worthiness of new customers before initial orders are accepted.

In addition, accounts receivable aging reports are evaluated for existing customers. Late payment or nonpayment can affect the terms of future purchases or, on rare occasions, cause PMW to refuse an order. Jose’s initial proposal to accept new customers via the Web was bad enough, but his new design to centralize all customer information is even worse, according to Fran.

**Round 3—Maybe I Should Understand the Question First**

The next day Jose is updating his resume and browsing monster.com. Mike drops by and asks for some time to discuss the e-Business project and the directory proposal. He makes several observations and recommendations.

1. The problems that Jose has encountered are a mix of technical and business problems, and therefore will require a mix of technical and business skills to solve. The approach to date has been solely technical and will not be successful, regardless of its technical merits.

2. Some of the problems Jose has encountered, such as lack of timely synchronization of employee data between the HR system and the internal e-mail system, have existed for years. The e-Business initiative did not cause these problems but could motivate people to help fix them.
3. Jose’s design proposal to deploy a new LDAP-based directory service, regardless of the feedback he’s received so far, is excellent. It makes perfect sense for new applications such as the e-Business project.

4. Jose must recognize that the e-Business application can neither stand on its own with no connection to existing data about employees and customers nor can it take over ownership of the data from existing implementations. An approach must be found that supports a robust e-Business solution without disrupting existing applications and ignoring or violating valid business policies. It must be evolutionary not revolutionary.

5. Existing organizations will continue to own and control permissions for employees and customer information. This implementation is based on valid business policies. The new e-Business application does need read access to some of the data and other permissions to other elements. Data modeling must be done to sort it all out by defining the semantics, syntax, policy, and permissions for each data object and attribute. Mike offers to help with the data analysis since the work will be similar to a recent systems management project PMW conducted to rationalize system-level data across heterogeneous platforms based on the distributed management task force (DMTF) model.

6. Existing applications will continue to access employee and customer data using existing program interfaces. Even if PMW endorses the concept of an enterprise directory system, there will be a lengthy staged migration for those applications that can be modified, and exceptions for systems that cannot change, such as the ISV e-mail package.

7. Mike feels that the business/political issues can be solved, but only if the existing organizations are involved in the solution. He recommends that Jose team with his IT group, Graham, Andre, and Fran on data modeling, and work out a migration strategy together rather than ask for their approval only at its completion.

**Round 4— Metadirectory**

Jose realizes that his project won’t succeed unless he accepts most of Mike’s recommendations. He gets agreement from the other organizations to work on a joint proposal.

The work starts slowly; each organization is willing to integrate systems and share common data—as long as it owns it and controls access. Mike points out that this is a common stumbling block. Twenty years ago a similar battle was waged over ownership of product part numbers between manufacturing and distribution when PMW computerized materials requirements planning (MRP). And just as the MRP issues were worked out by a combination of data modeling and business policy decisions, the ownership of employee and customer information is eventually resolved.

The final design proposal addresses the issues that were identified during the initial reviews and provides the new e-Business applications with the data it needs to operate.
Ownership / Access Permission

- The HR Department will continue to be responsible for creating and deleting employee records; that is, it will “own” the data. HR will continue to locate employee information in a separate database with few authorized users. Each employee entry has 20 different attributes. HR will allow six attributes that are not sensitive to be used by other applications. The usage permissions are read-only. The internal e-mail system will use all six and the Web white pages address book will use three.
- The Web white pages address book used by the e-Business e-mail application will list only those PMW employees that support customers. This is a small subset of the employees that use the internal e-mail system.
- The Customer Accounts Department will continue to be responsible for creating and deleting the customer information records; that is, it will own the data. It will allow other applications to use the information. The usage permissions are read-only.
- A new type of customer, “Internet customer,” can purchase PMW consumer-oriented products from the e-Business site using a credit card. No vetting is required by the Customer Accounts Department. The e-Business payment server will support auto-registration and authorize purchases for Internet customers. A semi-automated process, similar to the existing customer creation process used by the warehouse clerks, will be available for new commercial customers who wish to use the PMW e-Business Web.

Strategic Direction—Enterprise Directory Service and Migration

- PMW’s strategic direction for directory services is an LDAP-compliant directory service. New applications will use this service. Most existing applications will migrate to the LDAP directory over the next two years.
- Information required by multiple applications such as employee information and customer information will be located in the directory service based on business policy decisions. For example, employee attributes that are defined by HR as sensitive will not be located in the directory. The remainder of the employee data will be located in the directory service.

Tactical Direction—Synchronization

- In addition to being stored in the enterprise directory service, the data will also be synchronized between the directory service and other application-specific or existing system directories as needed. The need to synchronize among multiple application-based directories will decline over time as applications move to use the LDAP directory service, but it will not end. Some systems, such as the HR system and the enterprise e-mail system, will not use the directory service directly, due to the sensitivity of the data and the lack of a LDAP API, respectively.
- Synchronization of data across the enterprise will be automated to increase data integrity.
- Employee additions and deletions will be synchronized as soon as possible. Commercial customer records will be synchronized once a day.

**Implementation—Metadirectory** The technology chosen to implement this solution will be a metadirectory, which will provide

1. the connectors required to integrate with the various existing directory systems
2. the administration function to define the ownership and permissions model for the objects and attributes
3. the runtime join engine to update the directory service and define the direction and schedule for synchronization (See Figure 29.4.)

---

**E-PMW e-business Directory Integration Synchronization**

![Diagram of E-PMW e-business Directory Integration Synchronization](image_url)

**Figure AC.4 Synchronization**
Summary

This case study illustrates the metadirectory conundrum. The decision to deploy a metadirectory product is neither easy nor straightforward. Often it is reached at the end of an iterative, lengthy, and indirect process driven by new business opportunities that have, at least at the outset, little to do with directory services or metadirectory technology.

A successful metadirectory design must focus on the business and policy issues involved in sharing information across the enterprise, in addition to the technical architecture. Architectural purity must be tempered by organizational reality. Past experiences with data modeling and data integration projects such as MRP and DMTF can be applied to new projects based on open system standards such as LDAP directory services.

Finally, even where convergence onto an enterprise directory service is a strategic direction, valid business requirements for data synchronization among existing stores and other capabilities offered by metadirectory products will continue to exist.
This section contains technical appendices, sources for further information, a glossary, and an annotated bibliography. If you are new to directory programming, you will find it beneficial to browse Appendices A and B before getting too far into the book, but certainly before reading Part 2.
Appendix A
LDAP C API

This appendix provides a short introduction to lightweight directory access protocol (LDAP), along with a description of the underlying directory data model. One of the more common methods of using LDAP is through a set of C language callable programming interfaces, usually called the LDAP C API. To illustrate the use of the LDAP C API, a C++ language wrapper for these interfaces is developed. This C++ wrapper, the full source for which can be found on the accompanying CD, can be used to encapsulate the more complicated LDAP C API calls.

LDAP

Lightweight directory access protocol (LDAP) is a protocol for accessing a directory service. In its most basic form, LDAP describes a wire protocol (in the form of byte streamed data structures) for searching, comparing, and modifying information in a directory service. The definition of this protocol is important in order to separate directory exploiters from directory implementers, and allows both to work in parallel. In defining a protocol, the implementation of the client and server need not come from the same vendor. The wire protocol allows implementations from multiple software vendors to interoperate.

While many directory services have been defined in the past, the definition of LDAP, the access protocol, has placed the burden of directory interoperability on the providers of the directory, not on the exploiters of the directory. This allows directory exploiters to concentrate on their use of the directory, not on which directory services should be supported. Thus, the definition of LDAP has paved the way for directory exploitation by many applications that had not considered the benefits of exploiting a directory service in the past.

The LDAP protocol is a client/server protocol in which a client makes a request that is responded to by a server. For each request, the server responds with one or more response messages. A client can make multiple requests before receiving the first response from the server. In this sense, the LDAP protocol is asynchronous—
requests can be made without waiting for a response from the server. Each unique request sent from a client to a server is assigned a unique identifier. This identifier is used in all responses from the server that are sent to respond to the client’s request.

Seven operations are defined in LDAP: bind, add, modify, delete, modify name, search, and compare. All except the search operation return a single result message. The result message indicates whether or not an error occurred. The search operation can return multiple search result messages and multiple search referral messages, in addition to a single result message indicating whether or not an error occurred. Search result messages return information from the directory service.

Search referral messages contain information that a client can use to find more information that might match the original client search request. The search referral message contains the location of another server that can be contacted by the client in order to obtain more information matching the search request. It is the responsibility of the client to act on the search referral information by opening a separate connection to the referred-to server and re-sending the request to the server.

The client can be returned a referral indication in the case of other operations as well. In this case, the result message for the request contains the referral information. The use of referrals allows clients to initiate a request to one server and allows the server to indicate to the client the proper server to which to send the request.

Fundamental to all of the information sent to and received from the server over LDAP is the assumption that the data model of the underlying directory service is based on the X.500 directory model. The X.500 directory model is described in ITU recommendation X.501, summarized in the following section.

The Directory Data Model

The LDAP protocol provides a mechanism for performing a set of operations against a directory service. While the implementation of the directory service is not fixed by LDAP, the generic data model of the information stored in the directory service is fixed. The generic data model allows for customized information formats built from the generic data model. The data model is based on the X.500 directory model, described in ITU recommendation X.501. As described in X.501, the X.500 directory model is quite complicated. LDAP simplifies the interaction with this data model by using character strings to represent many aspects of the directory model.

The X.501 data model describes the directory as a hierarchy of entries. Within each node in the hierarchy (each entry), a set of attributes can be stored. Each attribute can be assigned one or more values. Attributes that are defined to contain multiple values are called multi-valued attributes; attributes that are defined to contain a sin-


single value are called single-valued attributes. In LDAP, attributes are represented with
a character string name. In most cases, string forms of attribute values are used
instead of complicated binary encodings. Generic binary attribute values are allowed
as well, so that virtually any data can be stored as an attribute value. Each attribute
has a syntax. Syntaxes define the structure of the attribute values for the attribute.
Examples of syntaxes are IA5String (a character string consisting only of 7-bit ASCII
characters), binary, integer, and GeneralizedTime.

In addition to having attributes, each entry has a name. The name of an entry is built
from a set of attribute-value pairs from within the entry, coupled with the name of
the parent entry in the hierarchy of entries. In this way, full names for each entry in
the hierarchy can be constructed. The full name for an entry is called its distinguished
name. The portion of the distinguished name that is made up of the attribute-value
pairs from the entry is called the entry’s relative distinguished name. All LDAP opera-
tions use the character string from of the entry’s distinguished name to indicate the
name of the entry in the directory hierarchy to which the operation applies.

Names of entries are usually made up of a small set of well-known attributes. While
not a requirement, names are usually made up of the following attributes: country
(abbreviated as c), organization (o), organizationalUnit (ou), and commonName (cn).
Other attributes that are commonly used to build names are domainComponent (dc)
and domain. These two attributes are used to build hierarchies of entries in the
directory that mimic names used by the domain name service (DNS). A very com-
mon structure is for an organization to use o=orgName as the root entry of its direc-
tory hierarchy. Under organizations are placed organizational units, and under
organizational units are placed entries that represent people in the organization.
CommonName attributes are used to name people within organizational units. Put
together, this naming format builds names that have the form cn=John Doe,
ou=northeast region, ou=sales, o=widgets, Inc. If domain component format
names are used, the name of an entry representing a person could look like cn=John
Doe, dc=northbranch, dc=widgets, dc=com.

One special attribute contained in every directory entry is called the objectClass of
the entry. The objectClass attribute, a multi-valued attribute, defines the structure
of the entry. The structure of the entry defines the set of attributes that must be
present in the entry and the set of attributes that may be present in the entry.

The format of entries and the format of attributes comprise the schema for entries
and attributes in the directory. The schema for directory entries and attributes is usu-
ally defined and stored in the directory server, and entries are checked by the server
during processing to ensure that the entry conforms to the schema defined in the
server. Applications that exploit the directory typically provide additional schema
information that must be installed in the server.
The LDAP add operation is used to add a new entry in the directory hierarchy. The LDAP delete operation is used to delete an entry from the directory hierarchy. Only entries that have no children can be deleted. The LDAP modify operation allows the attributes and values in an entry to be changed (added, modified, or deleted). The LDAP modify name operation allows the name of an entry to be changed. The LDAP search operation retrieves attributes and values for entries that match the search criteria sent in the search request. The LDAP compare operation checks to see whether a specific attribute-value pair exists in an entry. The LDAP bind operation allows a client to authenticate itself to the directory server. Once the client’s identity is known, the directory server may allow it more or less access to information stored in the directory.

Because of this generic directory model—a hierarchy of entries that contain attributes with one or more values—the low-level interface for searching and updating this information need not necessarily understand the specific schema for information stored in the directory. This allows the interface to be generic and usable across exploiting applications.

The LDAP C API

The LDAP protocol can be used in a number of ways. Programs can be written directly to the LDAP protocol, opening TCP/IP sockets, formatting protocol elements, and processing returned messages. This, however, is quite tedious, time-consuming, and error-prone. Instead, most LDAP exploiting applications make use of a programming interface to interact with directory servers that communicate with clients using LDAP. There are a number of programming interfaces available, ranging from the procedural-based C APIs to object-oriented interfaces such as the ActiveX Directory Services Interface (ADSI) and Java Naming and Directory Interface (JNDI).

The LDAP C API is a relatively simplistic set of interfaces offering C language function calls that match the LDAP protocol elements almost one-to-one. This mapping allows complex interactions to be performed with servers that support LDAP and provides access to all aspects of the protocol. The LDAP C API can, at first, seem hard to understand, but with a bit of guidance, it can be used to perform complex operations with information in the directory.

LDAP Initialization and Termination

Like most programming interfaces, the LDAP C API must be initialized. The result of initialization is that an LDAP handle, usually referred to as the ld, is returned. While
the contents of the data pointed to by the ld are hidden to users of the API, the ld contains all the state information necessary for the LDAP C API to submit LDAP requests and receive LDAP responses. Items of information contained in the ld include the set of TCP/IP socket connections that have been opened for communicating with directory servers (referrals chasing may open up additional connections besides the primary connection to a server). Also, outstanding LDAP responses that have been received over the protocol but not yet requested by the application are held inside the ld.

Applications can have multiple LDAP handles opened at one time. It is recommended that a single ld be used to communicate with given directory server, because each ld represents a separate TCP/IP connection to the server. Operations sent over separate lds are handled independently by both directory servers and the LDAP C API. While not all LDAP C API calls require an ld, any calls that represent LDAP protocol operations do require the ld. The LDAP C API calls that do not require an ld operate on data that has been returned by the directory server to the client based on a previous client request.

In order to use the LDAP C API, the structures and functions that make up the API must be available for the C compiler to check. All of this information is contained in the ldap.h header file. This header file is usually installed into the common search locations for C language code compilation, so including this file should be done with a line like

```c
#include <ldap.h>
```

added to the top of the C language source file in which LDAP function calls will be made.

**Getting an LDAP Handle**

To obtain an LDAP handle ld, call either the ldap_open() or ldap_init() functions in the LDAP C API. The difference between the two APIs is that the ldap_open() function allocates and initializes an ld and opens a socket connection with the directory server. The ldap_init() function only allocates and initializes the ld. An example of calling the ldap_init() function is

```c
ld = ldap_init( "ldap.ibm.com", LDAP_PORT );
```

In the code example, the TCP/IP host name ldap.ibm.com is used to connect to the IP address corresponding to that name. The LDAP_PORT identifier is defined in the ldap.h header file as the default port assigned to the LDAP protocol, 389. The host name parameter on the ldap_init() and ldap_open() calls can also be specified in hostname:port format, allowing the host and port to be specified together. In this
In order to close all socket connections that are open to carry requests made against the LDAP handle, the `ldap_unbind()` function call is used. The `ldap_unbind()` function closes all open socket connections, frees outstanding unretrieved messages held under the ld, and deallocates the LDAP handle. An example of calling `ldap_unbind()` is

```c
rc = ldap_unbind( ld );
```

**LDAP Results**

The server responds to every LDAP operation with an LDAP result message. For most operations, the LDAP result message is the only returned information. For LDAP search operations, the LDAP result message is the last message returned by the directory server. Each LDAP operation is assigned a message identifier by the client. The server responses to the LDAP operation will all have the same message identifier as the original request. This allows the client to correlate server responses with outstanding client requests.

Besides the message identifier, each LDAP result contains an error code and, optionally, a message string that is sent by the server. The message string is a variable length field. The content of the message string is server dependent and not required to be set by the server. The intent of the message string is to allow the server to provide more descriptive information about an error than the LDAP error code can provide. LDAP results are received by calling

```c
rc = ldap_result( ld, msgid, 1, timeout, &result );
rc = ldap_parse_result( ld, result, &errorCode, &matched, &errorMessage, &referrals, &serverControls, 1 );
```

The parameters of interest for our purpose are the input message identifier (`msgid`) and output LDAP result (`result`) on the `ldap_result()` function call. The LDAP result is then passed as input to the `ldap_parse_result()` function, which parses the returned protocol message and returns the LDAP error code (`errorCode`), matched string (`matched`), and server error message (`errorMessage`) values. The `errorCode` is an integer, while the `matched` and `errorMessage` values are character strings. The character strings should be deallocated by calling

```c
ldap_memfree( matched );
ldap_memfree( errorMessage );
```

Handling the results returned from LDAP search operations is more complex. This is covered in the section on LDAP search.
LDAP Bind

While not required when using the LDAP Version 3 protocol, depending on the security levels in effect in the directory server, identifying the user of a client program to the directory server may be necessary in order to perform some operations against the directory. The LDAP bind operation is used to identify and authenticate a client user to the directory server. The term bind is over-used in the computing industry. In this case, the term bind means identifying and authenticating to the directory server. The LDAP bind operation requires a bind type, a distinguished name (name of an entry in the directory) and a credential. For the LDAP_AUTH_SIMPLE bind type, the credential is a password value. An example of calling `ldap_bind()` is

```c
msgid = ldap_bind( ld, dn, creds, LDAP_AUTH_SIMPLE );
```

In this example, the asynchronous form of invoking an LDAP operation through the LDAP C API is used. Each of the basic LDAP operations can be invoked in asynchronous or synchronous mode. When asynchronous mode is used, the returned `msgid` value must be specified on a call to `ldap_result()` in order to retrieve the result of the operation. An example of calling `ldap_result()` to get the result of the bind operation is

```c
rc = ldap_result( ld, msgid, 1, timeout, &result);
```

The synchronous form of the LDAP bind operation has a slightly different name, `ldap_bind_s()`, and returns a different set of parameters. The synchronous forms of the LDAP operations are generally more convenient, since the synchronous forms perform the `ldap_result()` processing on the caller's behalf. An example of invoking the “synchronous” LDAP bind operation is

```c
rc = ldap_bind_s( ld, dn, creds, LDAP_AUTH_SIMPLE );
```

If successful, the bind distinguished name is used by the directory server when it determines whether or not operations can be performed as well as whether or not entry information can be returned to the client. The bind distinguished name is used until the next LDAP bind operation is performed by the client or the connection to the server is stopped by the `ldap_unbind()` function.

LDAP Search and Compare

The LDAP search operation is used to retrieve information from the directory server. The LDAP compare operation is used to determine whether or not a specific entry in the directory contains a specific attribute and value. In the LDAP compare operation, only a true/false answer is returned by the directory server. Since the LDAP compare operation only returns a true/false result, invoking the LDAP compare operation is
much easier than invoking the LDAP search operation. An example of calling 
ldap_compare_s() is

dn = "cn=John Doe, ou=northeast region, ou=sales, o=widgets";
attrName = "sn";
value = "Doe";
rc = ldap_compare_s( ld, dn, attrName, value );

The return code will be set to an error code or to LDAP_COMPARE_TRUE or
LDAP_COMPARE_FALSE, depending on whether or not the entry with distinguished
name, dn, contained attribute, attrName, with an attribute value, value.

The LDAP search operation, while relatively easy to invoke, returns results that are
somewhat complicated to parse. In fact, parsing search results information is the
most complicated part of using the LDAP C API. The LDAPConnection C++ class
presented at the end of this appendix provides a method that parses through the
LDAP search result message and builds a DirEntry C++ class instance containing all
of the returned information. An example of calling ldap_search_s() is

baseDn = "ou=sales, o=widgets";
rc = ldap_search_s ( ld, baseDn, LDAP_SCOPE_SUBTREE,
       "(objectClass=*)", NULL, 0, &result );

The search in this example returns all entries that are found below the entry with dis-
tinguished name baseDn. The synchronous form of the LDAP search operation was
used and the set of results is returned in the result parameter. To walk through the
results that are returned, the following general algorithm is usually used.

for ( msg = ldap_first_entry( ld, result);
msg != NULL;
    msg = ldap_next_entry( ld, msg ) ) {
    for ( attrName = ldap_first_attribute( ld, msg, &ber );
        attrName != NULL;
        attrName = ldap_next_attribute( ld, msg, ber ) ) {
        attrVals = ldap_get_values( ld, msg, attrName );
        for ( i=0; attrVals[i] != NULL; i++ ) {
            /*
               * work with the attribute values here
               */
        }
        ldap_value_free( attrVals );
        ldap_memfree( attrName );
    }
}
ldap_msgfree( result );
The algorithm presented here for parsing through the returned search results allows each attribute value for each entry to be retrieved from the LDAP search results. The outer loop iterates through the results of the search. One result message is returned for each directory entry returned from the LDAP search operation. The second loop works through each attribute that is returned in the entry. The third, innermost, loop works through every attribute value that is returned for each attribute in each entry.

**LDAP Add, Modify, and Delete**

Of the add, modify, and delete LDAP operations, the LDAP delete operation is the easiest to use. This operation simply takes the distinguished name of the entry to delete as input. Only directory entries that have no entries below them in the directory hierarchy, so-called leaf entries, can be deleted from the directory hierarchy. An example of calling the `ldap_delete_s()` function is:

```c
dn = "cn=John Doe, ou=northeast region, ou=sales, o=widgets";
rc = ldap_delete_s( ld, dn );
```

This invokes the synchronous form of the LDAP delete API and deletes the entry with distinguished name, `dn`, from the directory hierarchy.

The LDAP add operation is invoked by calling the `ldap_add()` or `ldap_add_s()` LDAP C API. While the `ldap_add_s()` function has only a few input parameters, setting up one of the input parameters is somewhat complex: the LDAPMod array must be set up prior to invoking the APIs. An example of invoking the `ldap_add_s()` function is:

```
LDAPMod attr0, attr1, attr2, attr3;
LDAPMod *attrs [] = { NULL, NULL, NULL, NULL, NULL };
char *objectClassVals[] = { "inetOrgPerson", "person", "top", NULL };
char *cnVals[] = { "Jane Doe", "Janet Doe", "Janice Doe", NULL };
char *snVals[] = { "Doe", NULL };
char *telephoneNumberVals[] = { "555.555.1234", NULL };
attr0.mod_op = LDAP_MOD_ADD;
attr0.mod_type = "objectClass";
attr0.mod_values = objectClassVals;
attrs[0] = &attr0;
attr1.mod_op = LDAP_MOD_ADD;
attr1.mod_type = "cn";
attr1.mod_values = cnVals;
attrs[1] = &attr1;
attr2.mod_op = LDAP_MOD_ADD;
```
attr2.mod_type = "sn";
attr2.mod_values = snVals;
attrs[2] = &attr2;
attr3.mod_op = LDAP_MOD_ADD;
attr3.mod_type = "telephoneNumber";
attr3.mod_values = telephoneNumberVals;
attrs[3] = &attr3;
dn = "cn=Jane Doe, ou=northeast region, ou=sales, o=widgets";
rc = ldap_add_s( ld, dn, attrs );

As shown in the example, the complexity in calling the `ldap_add_s()` function lies in setting up the attributes and values to be added as attributes in the new entry in the hierarchy.

The `ldap_modify_s()` programming API takes similar arguments to the `ldap_add_s()` API. However, in the case of the modify operation, the `mod_op` field of the `LDAPMod` structure contains values indicating whether the values provided should be added, replaced, or deleted. As a special case, if no attribute values are provided in the `LDAPMod` structure, and the `mod_op` field is set to `LDAP_MOD_DELETE`, all attribute values, and hence the whole attribute, are deleted from the entry being modified. An example of calling `ldap_modify_s()` is

```c
LDAPMod mod0, mod1, mod2;
LDAPMod *mods[] = { NULL, NULL, NULL, NULL };
char *cnDelAttrs[] = { "Janet Doe", NULL }; char *cnAddAttrs[] = { "Jane T. Doe", "Janice T. Doe", NULL };
mod0.mod_op = LDAP_MOD_DELETE;
mod0.mod_type = "cn";
mod0.mod_values = cnDelAttrs;
mods[0] = &mod0;
mod1.mod_op = LDAP_MOD_ADD;
mod1.mod_type = "cn";
mod1.mod_values = cnAddAttrs;
mods[1] = &mod1;
mod2.mod_op = LDAP_MOD_DELETE;
mod2.mod_type = "telephoneNumber";
mod2.mod_values = NULL;
mods[2] = &mod2;
dn = "cn=Jane Doe, ou=northeast region, ou=sales, o=widgets";
rc = ldap_modify_s( ld, dn, mods );
```

In this example, the `commonName (cn)` attribute is updated to delete the value “Janet Doe” and to add two additional values; “Jane T. Doe” and “Janice T. Doe.” In addition, the `telephoneNumber` attribute is removed from the entry. As the example shows, multiple updates can be made to an entry within the same LDAP modify operation.
LDAP Modify Name

Because some of the attributes that are part of entries in the directory hierarchy are used in the name of the entries, changing these attribute values requires special processing. The LDAP protocol contains a special operation called modify name for handling this situation. The modify name operation acts like a rename function but, prior to LDAP Version 3, only allowed the attributes within the entry to be changed. Thus, the entry could not be moved to reside under another entry in the directory hierarchy. LDAP Version 3 added protocol elements for handling moving entries. An example of renaming an entry in the directory without changing the parent of the entry is

```c
oldDn = "cn=Jane Doe, ou=northeast region, ou=sales, o=widgets";
newRdn = "cn=Jessica Doe";
rc = ldap_modrdn_s( ld, oldDn, newRdn, 0 );
```

This operation would change the name of the entry to `cn=Jessica Doe, ou=northeast region, ou=sales, o=widgets`, and the `cn` attribute of the entry to contain an additional value, Jessica Doe. The third parameter (specified as 0 in the example) indicates whether or not the old naming attribute should be deleted.

The functions presented here form the basic set of LDAP C operations. More complicated functions allow the modification of certain properties of the C language client, such as whether or not to follow referrals automatically and whether or not to trace operations performed. In addition, the LDAP Version 3 protocol definition introduces the concept of client and server controls, which can be added to protocol operations. The LDAP C API provides access to setting these controls on operations that are performed. The CD contains a complete LDAP C API reference including information about LDAP controls.

The LDAPConnection C++ Class

The source code on the CD contains an implementation of a C++ class that represents an LDAP connection. This C++ wrapper for the LDAP handle contains methods corresponding to all of the primitive LDAP protocol operations. In addition, the add and search methods accept and produce, respectively, a `DirEntry` C++ class. The `DirEntry` C++ class provides an easier interface for building new entries to be added to the directory as well as parsing search results from searches made against the directory. For modifications of existing entries, a `DirModifications` C++ class is used.

While not all of the examples in this book make use of this `LDAPConnection` C++ class, the class may be useful in some environments and is available for use in C++ programs.
Summary

The LDAP C API is a relatively easy-to-use programming interface for accessing and updating a directory service using the LDAP protocol. Most of the C language APIs map to the LDAP wire protocol elements. Thus, if you come armed with an understanding of the LDAP protocol, using the LDAP C API is straightforward. For encapsulation and a level of isolation from the rest of a project’s source code, the LDAPConnection C++ class, provided on the CD, provides a granular yet simple-to-understand C++ interface to a directory service.
In contrast to LDAP, the Java Naming and Directory Interface (JNDI) defines a framework for multiple directory systems. This framework allows multiple directory services, with different naming and data models, to be accessed through a common set of programming interfaces. Examples of directory services and naming models that might be accessible through JNDI interfaces include Object Management Group (OMG) Common Object Services Naming (COSNaming), domain name service (DNS), and LDAP-accessible directory servers.

One of the more common uses of JNDI has been to access a directory service using the LDAP protocol. In this appendix, the same operations that were described in Appendix A will be described, with examples on how to invoke these functions over the LDAP protocol using the JNDI programming interface.

This appendix only covers JNDI accessing of a directory service using the LDAP protocol. JNDI covers a much broader scope than just accessing a directory service using the LDAP protocol. The service provider framework part of JNDI can be used to provide access to other types of information. Further, JNDI converges the concepts of distributed object naming and lookup with access to information in multiple directory services.

The complete javadoc for Java Naming and Directory Interface Version 1.2.1 can be found on the CD.

**Using JNDI to Access LDAP Directories**

JNDI is a framework for accessing multiple directory services. To implement this framework, two Java interfaces have been defined, `Context`, and `DirContext`. Different service providers supply directory-specific service provider implementations that instantiate the objects that implement the `Context` and `DirContext` interfaces. Because these objects are manipulated only through the `Context` and `DirContext` interfaces, many of the implementation specifics of the particular directory service...
implementation are hidden from the Java program. This allows directory-service independent applications to be written.

Accessing directory services with the LDAP protocol is done by using a JNDI LDAP service provider to instantiate the objects that implement the Context and DirContext interfaces. The object instances created by the JNDI LDAP service provider use the LDAP protocol to access and update information in the directory service.

The Context interface defines a set of methods that are closely aligned with storing information about a distributed object instance for later lookup and use by a client object in need of connecting to the distributed object. These methods are based on the methods defined by the OMG COSNaming standard.

The DirContext interface defines a set of methods that are closely aligned with storing general information in a directory service, based on a model of attribute-value pairs. These methods are closely related to the LDAP programming APIs, including methods for adding entries, modifying entries, and deleting entries in a directory service.

**JNDI Initialization and Termination**

Using JNDI to initiate and terminate connections to LDAP servers is very different from using the LDAP C API. There is no concept in JNDI of an LDAP handle, which represents the connection to the LDAP server, as there is in the LDAP C API. Rather, an LDAP connection is represented with an instance of a class that implements the DirContext interface.

Initializing a connection to an LDAP server using the JNDI LDAP service provider involves setting some Java properties and creating an instance of a class that implements the DirContext interface. Terminating the connection is performed implicitly as part of program termination, garbage collection, or finalization. All of the information that was required to be set for initialization of the LDAP C API must be supplied in Java properties, which the JNDI LDAP service provider reads.

Two Java properties affect initialization of the object that implements the DirContext interface:

- `java.naming.factory.initial`
- `javax.naming.provider.url`

These Java properties must be set prior to creating an object that implements the DirContext instance. The first property indicates what Java class should be used for instantiating the first (initial) object that implements the DirContext interface. The
second property indicates the DNS host name and port number of the LDAP server that is to be contacted.

**Getting an LDAP DirContext**

As described, creating an instance of a class that implements the DirContext interface involves setting up a number of Java properties. After these Java properties are set up, an object that implements the DirContext interface is created as an instance of an InitialDirContext class. An InitialDirContext is a special form of a DirContext interface that contains all the methods a DirContext interface contains. From an LDAP perspective, the returned instance of an InitialDirContext class can be viewed somewhat like the LDAP handle in the LDAP C API.

An example of creating an initial object that implements the DirContext interface is

```java
Properties props = new Properties();
props.put("java.naming.factory.initial", "com.ibm.jndi.LDAPCtxFactory");
props.put("java.naming.provider.url", "ldap://ldap.ibm.com:389");
DirContext ctx = new InitialDirContext( props );
```

The `java.naming.factory.initial` property must be set to the name of the Java class that will instantiate a JNDI service provider. In the example, the LDAPCtxFactory class will create instances of objects that support the DirContext interface. The `java.naming.provider.url` property tells the object instance that implements the DirContext interface which TCP/IP host name and port number to use in opening an LDAP connection.

When the constructor for the InitialDirContext class is invoked, a new instance of an object that implements the DirContext interface is created. Note that this constructor can throw a NamingException that must be caught by the calling method. When no Java properties are supplied in the constructor arguments for InitialDirContext(), the properties for the system are used (System.getProperties()).

Once an instance of an object that implements the DirContext interface is created, operations can be performed against it that are similar to operations that can be performed by the LDAP C API.

**JNDI Results**

JNDI makes use of Java exceptions for returning results to callers of the DirContext interface. Successful operations result in no exceptions while exceptions are thrown.
for all error conditions. Handling results, then, is a matter of setting up a try/catch block around the operation to be performed. An example of this is

```java
try {
    SearchResult result = ctx.search( "ou=sales, o=widgets",
            "(objectclass=*)", srchCtls);
} catch( NamingException ne ) {
    System.out.println( ne );
    System.exit(-1);
}
```

All exceptions thrown by JNDI inherit from `NamingException`. Descriptive information about the error can be retrieved by invoking methods on the `NamingException`. Information such as an explanation of the error or the portion of the name that was successfully found during the operation can be retrieved. The example makes use of the `toString()` method of the `NamingException` to create a descriptive output message.

### JNDI Bind

Unlike the LDAP C API set, there is no explicit LDAP bind (meaning authenticate) operation defined in JNDI. To confuse the issue, the `Context` interface (and thus the `DirContext` interface) contains methods called `bind()` that perform a very different function than an LDAP bind. The `bind()` methods in the `Context` and `DirContext` interfaces correspond to the OMG COSNaming bind function, which attaches a name with a distributed object instance.

To allow for authenticated access and update to a directory service accessible over the LDAP protocol, additional Java properties are used. These Java properties provide the JNDI LDAP service provider with the information it needs to perform the LDAP bind operation. With the JNDI LDAP service provider from IBM, three Java properties are used for handling an LDAP simple bind.

- `java.naming.security.authentication`
- `java.naming.security.principal`
- `java.naming.security.credentials`

An example of using these Java properties is

```java
Properties props = new Properties();
props.put("java.naming.factory.initial", "com.ibm.jndi.LDAPCtxFactory");
props.put("java.naming.provider.url", "ldap://ldap.ibm.com:389");
props.put("java.naming.security.authentication", "simple");
```
In the example, the LDAP bind operation will flow to the directory server as part of setting up the DirContext instance.

### JNDI Search and Compare

In order to compare attributes in the directory service using JNDI, the DirContext search method must be used. A separate compare method is not defined as part of the DirContext interface. Rather, a base-level search, specifying only one attribute and value in the search filter, approximates the compare operation. An example of setting up a search in this fashion is:

```java
String name = "cn=John Doe, ou=northeast region, ou=sales, o=widgets";
String filter = "(sn=Doe)";
String retAttrs[] = { "dn" };
SearchControls srcChls = new SearchControls();
srcChls.setSearchScope( SearchControls.OBJECT_SCOPE );
srcChls.setReturningAttributes( retAttrs );
SearchResult result = ctx.search( name, filter, srcChls );
```

A small trick here minimizes the chance that any attributes are returned from the search operation (recall that an LDAP compare operation does not return any attribute values to the caller). The specification of `dn` as the attribute to return will result in no attributes returned, but the return code from the server will be based on the results of the search. This approximates the compare operation by using a base-level search operation.

Depending on the type of operation performed using the DirContext interfaces to access information accessible over LDAP, a different set of Java classes will be returned from the operation. The most complicated result returned is the information returned from a search operation. Results of read operations and list operations return subsets of the information returned from a search operation.

Two methods are defined in the DirContext interface that perform search operations. The getAttributes() method returns information from a single entry in the directory. This is similar to performing a base-level search on an entry. The returned value from the getAttributes() method is an Attributes class. The search() method can be set up to return information about a single entry, all entries one level below a given entry, or a whole subtree of entries below an entry. With this flexibility comes complexity in the handling of the information that is returned by the method. An example of performing a subtree search is:
String baseDn = "ou=sales, o=widgets";
SearchControls srchCtls = new SearchControls();
srchCtls.setSearchScope( SearchControls.SUBTREE_SCOPE );
NamingEnumeration results = ctx.search( baseDn, "(objectClass=*)", 
    srchCtls );

A search operation can result in multiple directory entries being returned to the 
calling application. To handle this situation, the returned data from the search 
operation is a Java class called a NamingEnumeration. NamingEnumeration 
instances represent the set of directory entries returned from the search. Each ele-
ment of the NamingEnumeration is a Java class called a SearchResult. Each 
SearchResult class contains a set of information stored in an Attributes class. The Attributes instance contains a set of Attribute class instances. The whole set of 
Attribute instances is returned as a NamingEnumeration. In this case, the Naming-
Enumeration elements are Attribute classes. Each Attribute instance contains 
multiple values that are returned as a NamingEnumeration. In this case, the Naming-
Enumeration elements are Java Object instances. An example of dealing with this 
complex result format is

```java
while ( results.hasMore() ) {
    SearchResult res = (SearchResult)results.next();
    Attributes attrs = res.getAttributes();
    NamingEnumeration attrsEnum = attrs.getAll();
    while ( attrsEnum.hasMore() ) {
        Attribute attr = (Attribute)attrsEnum.next();
        NamingEnumeration valuesEnum = attr.getAll();
        while ( valuesEnum.hasMore() ) {
            String value = (String)valuesEnum.next();
            // work with the attribute values here
            /*
            * System.out.println( "name: " + res.getName() +
            * " attr: " + attr.getID() +
            * " value: " + value );
            */
        }
    }
}
```

The algorithm presented here for parsing through the returned search results allows 
each attribute value for each entry to be retrieved. The outer loop iterates through 
the results of the search. One SearchResult is returned for each directory entry 
returned from the DirContext search method call. The second loop works through 
each attribute that is returned in the entry. The third, innermost, loop works through 
every attribute value that is returned for each attribute in each entry.
JNDI Add, Modify, and Delete

The DirContext interface supports adding, modifying, and deleting entries from the directory service. These operations are performed using the createSubcontext(), modifyAttributes(), and unbind() methods, respectively. These methods are defined by the DirContext and Context interfaces. Like the LDAP C API, the delete operation is the easiest to perform. An example of calling the unbind() method is

```java
String dn = "cn=John Doe, ou=northeast region, ou=sales, o=widgets";
ctx.unbind( dn );
```

The result of this operation is that the entry \texttt{cn=John Doe, ou=northeast region, ou=sales, o=widgets} is removed from the directory.

The createSubcontext() method is used to add new entries into the directory service when using the DirContext interface. Like the LDAP C API \texttt{ldap_add_s()} function, the createSubcontext() method takes only a few parameters. However, the parameters are somewhat complex to set up. An example of adding a new entry into the directory using createSubcontext() is

```java
BasicAttribute attr0 = new BasicAttribute( "objectClass" );
attr0.add( "inetOrgPerson" );
attr0.add( "person" );
attr0.add( "top" );
BasicAttribute attr1 = new BasicAttribute( "cn" );
attr1.add( "Jane Doe" );
attr1.add( "Janet Doe" );
attr1.add( "Janice Doe" );
BasicAttribute attr2 = new BasicAttribute( "sn" );
attr2.add( "Doe" );
BasicAttribute attr3 = new BasicAttribute( "telephoneNumber" );
attr3.add( "555.555.1234" );
BasicAttributes attrs = new BasicAttributes();
attrs.put( attr0 );
attrs.put( attr1 );
attrs.put( attr2 );
attrs.put( attr3 );
String dn = "cn=Jane Doe, ou=northeast region, ou=sales, o=widgets";
ctx.createSubcontext( dn, attrs );
```

In the example, a new entry named \texttt{cn=Jane Doe, ou=northeast region, ou=sales, o=widgets} is added to the directory. Most of the work in adding an entry to the directory is in setting up the BasicAttributes collection of attributes that comprise the information to be stored in the new directory entry.
Unlike with the LDAP C API, the method used to modify attributes in a directory entry is somewhat different than the createSubcontext() method. The modifyAttributes() method requires the distinguished name and an array of ModificationItem objects as input. Each element of the array of ModificationItem objects represents an attribute modification. An array is used to preserve the order in which the modifications are applied to the directory entry. An example of modifying an entry is

```java
BasicAttribute attr0 = new BasicAttribute( "cn" );
attr0.add( "Janet Doe" );
BasicAttribute attr1 = new BasicAttribute( "cn" );
attr1.add( "Jane T. Doe" );
attr1.add( "Janice T. Doe" );
BasicAttribute attr2 = new BasicAttribute( "telephoneNumber" );
ModificationItem mods[3];
mods[0] = new ModificationItem( DirContext.REMOVE_ATTRIBUTE, attr0 );
mods[1] = new ModificationItem( DirContext.ADD_ATTRIBUTE, attr1 );
mods[2] = new ModificationItem( DirContext.REMOVE_ATTRIBUTE, attr2 );
String dn = "cn=Jane Doe, ou=northeast region, ou=sales, o=widgets";
ctx.modifyAttributes( dn, mods );
```

In this example, the commonName (cn) attribute is updated to delete the value “Janet Doe” and add two additional values, “Jane T. Doe” and “Janice T. Doe”. In addition, the telephoneNumber attribute is removed from the entry. As the example shows, multiple updates can be made to an entry within the same DirContext modifyAttributes() invocation.

**JNDI Modify Name**

The DirContext interface supports renaming the left-most portion of an entry name using the rename() method. The rename() method is defined as part of the Context interface which DirContext inherits from. An example of renaming an entry is

```java
String oldDn = "cn=Jane Doe, ou=northeast region, ou=sales, o=widgets";
String newRdn = "cn=Jessica Doe";
ctx.rename( oldDn, newRdn );
```

The result of this operation is that the directory entry will now have the name cn=Jessica Doe, ou=northeast region, ou=sales, o=widgets.
Summary

This appendix serves as a brief introduction to using the JNDI interfaces for accessing a directory service. Most of the operations that can be performed using LDAP C APIs can be performed using JNDI DirContext interface methods coupled with a JNDI LDAP service provider. The JNDI programming interface allows access to directory services from the Java programming environment. JNDI allows multiple naming formats to be integrated to form a composite name space, and integrates distributed object naming methods (Context) with directory service access and update methods (DirContext). JNDI is a powerful programming tool for multi-platform, multi-service environments. Examples of using JNDI are provided on the CD. In the section for this appendix, an LDAPConnection Java class is provided which has a similar interface to the LDAPConnection C++ class provided on the CD for Appendix A.
Appendix C

ADSI

This appendix summarizes the Active Directory Services Interfaces (ADSI), a Microsoft API layer for accessing directory and other information.

ADSI is a component object model (COM) system for matching data sources to data clients. The matching layer is called the provider layer. An ADSI provider is a set of COM objects that translate a well-known set of interfaces into a request for some data source. For example, there is a provider for LDAP that translates the ADSI COM interfaces into an LDAP datastream on the wire that LDAP servers can understand. There is also a provider for Novell's NDS that translates these same well-known interfaces into a datastream on the wire that NDS can understand. Anyone can write a provider to go from the same Microsoft-defined ADSI COM interfaces to another data source. For example, IBM has written a provider for the IBM MQSeries queuing product that enables queues to be manipulated using ADSI. (See Figure C.1.)

Since the provider layer is exposed as COM interfaces, clients can program in any language supported by COM, such as C/C++ and Visual Basic. Requests are routed to ADSI providers by a router, and they flow from the provider through to data sources, such as Microsoft's Active Directory, other vendors' LDAP directories, IBM's MQSeries, and so on. It is up to the provider to find the data source (such as a directory) and build a request that the data source can understand (such as an LDAP wire flow).

There are many ways to access a directory, including ADSI. Some of these ways are shown, along with ADSI, in Figure C.2. One of the most popular methods for accessing directories is with the LDAP C API, which is also shown in Figure C.2 emanating from Microsoft's and IBM's LDAP C.
Figure C.1 ADSI providers translate client requests into something that a data source can understand.

Figure C.2 ADSI and other ways to access directories.

AD = Active directory
SAM = Security accounts manager
Using ADSI

Many Windows system administrators use Visual Basic or scripts to access data using ADSI. ADSI is particularly easy to use from Visual Basic. Objects in the Active Directory, for example, appear to Visual Basic as objects on which methods can be called. For example, in binding to a domain object (a Windows 2000 domain is a collection of Windows computers),

```vbscript
Set domain = GetObject("LDAP://DC=xyzxyz,DC=com")
```

“domain” is an object representing the Windows 2000 domain whose DNS name is xyzxyz.com. The call above will bind to an instance of the Windows 2000 Active Directory (an LDAP directory built in to Windows 2000). Using the domain object, we can create other objects within the domain. For example, the following creates a new organizational unit object called “EasternSales” in the xyzxyz.com domain.

```vbscript
ESOU = domain.Create("organizationalUnit",OU=EasternSales)
ESOU.Put "description","Eastern Sales Personnel"
ESOU.SetInfo
```

This code first creates an OU called EasternSales, then sets the description to “Eastern Sales Personnel,” and finally sets the information into the directory. Nothing is actually written to the directory until the set operation is invoked, so any number of put operations could have been batched into the one set operation.

ADSI and C/C++

Many programmers and administrators use C/C++ to build ADSI programs, instead of Visual Basic or other languages. ADSI is slightly different in C/C++ than in Visual Basic. Generally, C/C++ programming involves more knowledge of COM. For most C/C++ ADSI programming, one of the following five interfaces is used: IADSOpenDSObject, IADS, IADsContainer, IDirectorySearch, and IDirectoryObject (the “I” prefix is used to denote an interface, which may be composed of many methods).

In C/C++, pointers to these interfaces can be obtained by invoking the AdsGetObject function. AdsGetObject is similar to the GetObject previously shown for Visual Basic, except that AdsGetObject returns an interface pointer (IADS, for example). Once an interface pointer has been obtained onto a COM object, methods on that object can be called using the interface pointer. For example, Put and SetInfo are two of the methods on the IADS interface (recall that Put and SetInfo are shown in the Visual Basic sample above as well).
For More Information and Samples

For more information about ADSI, visit Microsoft’s Web site. A great deal of ADSI information is contained in the Microsoft Platform SDK, which can be obtained with a Microsoft Developer Network (MSDN) subscription or on-line.
Appendix D

WEB APIs and Protocols

The World Wide Web has progressed through several evolutionary stages. Much of the progress happened rapidly and concurrently. This appendix discusses the history of the Web, protocols, client APIs, and server APIs.

Projecting Static Brochures to Clients

The World Wide Web started as a way for users to find and view a static document page. It enabled fixed brochures organized within a connected Web site to be located and handed back to unsophisticated users of the Internet. This is still the most common client-server Web exchange today.

A Web server houses each document in a set of one or more flat files—usually more. A set of 30 files per page isn’t uncommon today. The server has no knowledge or interest in the content of each file. The URL addressing scheme encapsulates the information needed to locate the server and each requested file. The TCP/IP DNS directory is used to locate the server. TCP routers route the request and response packets. The Web protocol is called HTTP. The client Web browser has the knowledge to decode and render the base file on the client screen. During rendering it may discover more URLs or referenced files in the page, causing it to make more requests to the server to obtain all the files it needs to render a document page. Often, these multifile pages meld the work of graphic artists and textual authors into beautiful, albeit static, presentations. (See Figure D.1.)

The essence of a basic Web client-server interaction is as follows.

1. Universal resource locator, such as http://www.ibm.com/index.html.
2. Domain name system, a distributed database used to map names to IP addresses or vice versa. DNS is a type of directory.
3. Hypertext transfer protocol, a request-response protocol over a session socket.
1. A user requests a file through a Web browser using a URL.
2. The browser uses the host portion of the URL to locate the server.
3. The browser uses the protocol portion of the URL to talk to the located server. The protocol is usually HTTP, but it could be FTP or even LDAP.
4. A conversation starts with the browser sending the URI—the file request information—to the server.
5. The server maps the URI into a local directory and file name.
6. The server returns the file (or an error) over the open socket and then it closes the session.
7. The browser saves or renders the file based on its type. Often it is an HTML file. During rendering it may encounter a link reference URL. Such a URL would be used to start a new sequence at step 2. These requests may proceed in parallel over several sockets.

Notice that the server is stateless: It doesn’t concern itself with file content. Later, when we talk about server-side includes, the server rummages through the content. For now, the browser actually has the more complex task; the server is mainly concerned with scalability.

---

4. Universal resource identifier.
5. HTTP was later extended to keep the session if another response occurred soon enough. This is called HTTP keep-alive.
6. Hypertext markup language; used to define a page to a Web browser. Older Web servers knew nothing about HTML.
Web Protocols

The World Wide Web is built on hypertext transfer protocol, or HTTP. This simple protocol provides two-way data streams described by header records. A choice of exchange mode is specified in the header. Get and put are the most frequent modes. A TCP session is set up and destroyed for each two-way exchange of request and response data. This simple paradigm is detrimental to server and network performance. HTTP gained a feature called “persistent sessions” to reduce this problem. This enables a session to be re-used within a certain time-out period. Its use is optional, but it makes more of the bandwidth for content instead of session control.

Since the Web is a file projection mechanism, many other protocols are also used. For instance, it is possible to use an FTP server as a Web server, in which case the HTTP protocol isn’t used. The author did this in the early days of the Web, when a free Web server for NT 3.1 was not readily available. The URL protocol field must be specified as FTP instead of HTTP, and there is no default start page if no page is specified in the URL. The browser will display a directory tree of file links in that case. If an HTML file is chosen it will be rendered as if it came from an HTTP Web server. The lack of a default page and the fact that an FTP server has no program execution capability is the main deficiency. Many download pages reference an FTP server for efficient download of software. If an FTP server needs an identified login, the URL format provides for a user ID and password. These are sent in clear text, however. Normally, FTP is used to distribute fairly stable data such as downloads.

IETF RFC 997 describes a separate protocol, network news transfer protocol (NNTP). NNTP is used to distribute transient new format information over the Internet.

Gopher is a lightweight protocol similar to HTTP. It is included in every Web browser so that Gopher information may be browsed as part of the Web. Gopher menus are easily mapped to hypertext links. The number of Gopher sites has reduced in recent years.

This book is about directories, so it is nice that Web browsers know how to make directory queries using the LDAP protocol. LDAP search parameters are appended to the URL like the query parameters of an HTTP GET parameter string.7

The wire protocol for Microsoft’s COM model is called DCOM. It is based upon DCE RPC. ActiveX COM objects running in the browser may talk to COM+ business objects running in a Microsoft Web server independently of HTTP. The

---

7. For LDAP URL format, see http://lists.w3.org/Archives/Public/uri/1995Apr/0012.html or IETF RFC 2254.
COM+ business objects, in turn, usually communicate with a DBMS using COM-based OLEDB. DCOM is the protocol here.

Java applets may communicate with Java code running on the server using remote method invocation (RMI). Applets are restricted to communication with the same machine as the Web server, but a proxy Java application may talk to any node using any protocol on behalf of the client. Non-Java servers may be wired in through CORBA IIOP. Java allows RMI to be used over IIOP.

A common characteristic among DCOM, RMI, and RMI over IIOP on the Web is that an application is often set up with a client using HTTP but it uses one of the other protocols for its own communication. The server side of the application may use any protocol available to work with other servers on the client’s behalf.

The MIME protocol is used by the Web browser as a clue as to how to render a file, as covered in the next section.

Web Client APIs

Web clients use the file type extension returned from the server to know how to render a file. The type is matched against a table of MIME types in the browser. The table may indicate that an auxiliary application is be executed against the file. For instance, a .pdf file extension may cause the browser to invoke Adobe Acrobat to render the file. If the file type is not registered in the MIME table, the browser offers to download the file. This MIME mechanism may be considered a way to plug APIs into the browser.

No more than a couple of “Web years” elapsed before the static rendering engine of many browsers gained the ability to execute programs and objects that decorated the HTML tags.

Client-Side Scripting

The second evolutionary stage of the client side of the Web involved more sophisticated browsers that could execute scripts embedded in document pages. As a browser renders a page, it detects and stores these scripts. Routines may be attached to user interactions with form elements, or the page load itself may trigger a call to a script

8. Multipurpose Internet mail extensions, described in RFC 1341. MIME was conceived as a way to attach many types of data to a text e-mail message. The Web hijacked MIME as a file identification mechanism.
function. The page author decorates certain HTML tags with script references such as "onClick = some function." (See Code Example D.1.)

Such scripting led to some cute HTML pages. More importantly, it enabled form-based applications to cut down on round trips to the server. Web servers can execute programs for Web clients. Client-side scripting allows click-detection on images to be handled by the client instead of the server, as was done in the early Web years. Textual input may be validated in the client. A message box saying “You must enter your ZIP code” could pop up over a browser without involving the network or the server. This adds a little fat to the client, but the page source code resides on the server, so the version control sits in one place.

**Java Applets**

Java had its origins as Oak, an embedded system language. A set of Java classes today address embedded systems, but the larger Web world first saw Java in the form of Java applets, small object-oriented programs with a graphic user interface. Java applets are compiled to run in a well-defined abstract virtual machine. Thus they may run on any system that has a proper Java virtual machine available.

Applets plug into HTML pages using the APPLET tag or the OBJECT tag. A PARAMETER tag allows the HTML author to specify input parameters. The APPLET tag specifies a window rectangle on which to paint the applet UI. The

```html
<SCRIPT language=JavaScript>
  // Handle the "prev" button
  function HTMLPrev_OnClick()
  {
    if (--currentpix < 0)
      currentpix = maxpix - 1;
    setImage(currentpix);
  }
</SCRIPT>

...<INPUT TYPE="submit" VALUE="Previous" NAME="HTMLPrev" onClick="HTMLPrev_OnClick()">
```

**Code Example D.1**  Client-side scripting
browser runs the applet in a Java virtual machine within the browser address space. The applet interacts with the user. It may also interact with the Web server or a separate software server on the Web server machine using any protocol.

Tight security is built into Java applets. An applet cannot write to the client file system, nor can it operate with any server machine other than the Web server machine from which it was downloaded. These security policies may be modified by administrator action in later Java releases.

HTML scripting may interact with Java applets. This allows a Web page author to create the client side of an application from reusable generic applet tools, specifying custom behavior through parameters gained from scripting. This model is also used for ActiveX controls.

Java beans are a component model for Java classes. A bean has discoverable interfaces, properties, and events. A Java applet may be written as a Java bean. Tools such as IBM Visual Age for Java make Java bean applet creation easy.

ActiveX Controls

Microsoft ActiveX controls appeared in 1996. These plug into HTML pages using the OBJECT tag to identify the control and specify a rectangle for the control to render a UI. An ActiveX control is a COM component that has certain discoverable interfaces implemented. It is a dynamic link library that is downloaded from the Web server using an asynchronous COM moniker. The ActiveX control idea emerged from 16-bit Visual Basic VbX components used to construct monolithic VB applications in Windows 3.1. At first the 32-bit COM-based VbX successor was called an OLE control, or OcX. An OcX was a heavyweight control that implemented many COM interfaces.

In late 1995, after releasing Windows 95, Microsoft realized that the Web had to be part of its future. As part of a vast, 1,000-day change in direction, Microsoft refactored OLE. A lightweight OcX called an ActiveX control emerged that could be loaded asynchronously over HTTP. Many of the mandatory OcX interfaces became optional to ease the download footprint. The container, Internet Explorer 3, had to discover which interfaces existed and make up for the missing ones. Netscape for Windows gained the ability to run ActiveX controls only through a third-party plug-in.

One disadvantage to ActiveX controls is that since they are COM objects they are limited to the Win32 API. This confines their practical use to Windows platforms, a disadvantage on the Internet at large. Some sites detect the browser and ask the user’s permission to download ActiveX. ActiveX controls are used with some corporate intranets behind firewalls, where the browser type is known.
A second disadvantage is that Windows DLLs are being loaded into the client machine. To counter this problem, Microsoft has provided a certificate-based code-signing architecture where the signature is certified by a trusted certificate authority. This allows the client user to decide whether to trust the originator of the control, knowing that the control had to come from that origin. There were one or two successful attacks using ActiveX controls, but Microsoft soon supplied hot fixes.

One advantage of ActiveX controls is that they are COM objects that can speak to remote COM objects using DCOM, or Windows itself using COM, or any Windows API. This ability could enrich many enterprise applications.

Another advantage of ActiveX controls is that Microsoft has supplied tools within Visual Studio that makes their development quite easy. Among these tools are Visual Basic, Visual Java, and Visual C++ with MFC. The controls can be integrated with HTML using Visual Interdev. Interaction with the user is orchestrated through HTML scripting as it is with Java applets.

**Dynamic HTML with Cascading Style Sheets**

Major browser vendors imposed object models on their respective HTML rendering engines. Most of the document page, the current window, interactive events, and all of the HTML tags and attributes became part of this object model. This was, in turn, exposed to client-side scripting. The common name for this is dynamic HTML or DHTML. Cascading Style Sheets (CSS) technology allows control of page rendering by the page author.

When CSS was combined with DHTML and client-side scripting some rich effects became possible. Unfortunately, Microsoft Internet Explorer and Netscape Communicator provided different DHTML implementations. A common ad hoc scripting task—browser detection—was necessary for well-written rich Internet client scripting. The game of Frogger is an entertaining example.

The World Wide Web Consortium (W3C) regards Dynamic HTML as a vendor term for combining style sheets, scripting, and HTML to allow documents to be animated. The W3C has concentrated on a specification for a common document object model of the way that documents should be exposed to scripts. No new HTML tags are

---

10. See http://www.w3.org/Style/CSS/.
11. If you have Netscape 4.1 or greater, try Frogger at http://www.usagreen.org/frogger/cant-play.html.
12. See http://www.w3.org/.
13. See http://www.w3.org/DOM/.
involved. The specification is up for review, but no common document object model exists in any commercial browser.

**Web Server APIs**

It was recognized early on that a Web server is capable of executing applications on behalf of a client. The process of resolving client URLs requests to files that are projected back to the client is just one possible application that may be carried out on the client’s behalf. Today, advanced Web servers control stateful sessions with clients. These are application servers that may sit behind the traditional Web server. The application layer may allow a choice of Web servers to be used. IBM’s WebSphere Application Server is a Java-based application server that supports several choices of Web server.

**Common Gateway Interface**

The common gateway interface (CGI), appeared early. A CGI Web server is sensitized to certain logical directories requested in URIs. Remember that the URI is the logical file path extracted from the URL by the browser and that it is sent to the server with a request. Commonly, if the request is for a file in the cgi-bin directory, the server should execute the requested file instead of returning it. The HTTP protocol specifies how parameters and results are interchanged between client and server. The CGI application usually returns output to the client as HTML tags.

How are the input parameters created? They may be input as part of the URL syntax. This is part of the HTTP network protocol get operation. The get operation has length restrictions. It is normally used for specifying search arguments. Parameters are specified as name-value pairs.

Parameter name-value pairs may alternatively be passed within HTTP header records. This variation is called HTTP POST. There is no length restriction. How are these parameters input? That leads to the next question: How does the client interact with the CGI application? A client could append parameters to the URL: a question mark followed by name-value pairs. Special characters and blanks are encoded in an arcane syntax. Humans usually don’t type these URLs into a browser. Instead, an HTML form generates the parameters from HTML tags on a page. The page originates because the client requested it from the server. The user acts on fields in the page, then hits a “search” or “submit” button. The parameters are gathered by the browser and encoded into a GET URL or sent to a POST stream. The CGI application runs, acting on the parameters, and generates a response HTML page. That page could have another version of the HTML form on it. This flow cycle is shown in Figure D.2.
The HTTP and CGI specifications were designed to work with as many operating systems and languages as possible. For example, the CGI application obtains parameters deposited into environment variables by the Web server’s HTTP layer because environment variables are ubiquitous across operating systems. Three characteristics of CGI are as follows:

- A CGI program is executed as a separate process. A Unix Web server forks it and a Windows NT Web server carries out a CreateProcess API call.
- The lifetime of the CGI program is one client-server request-response exchange.
- Neither application execution state nor any notion of a client-server session exists across executions.

Imagine a Web server that is hit with thousands of CGI requests per hour. The process creation and destruction will cause the machine to hit the wall early.

**Improving CGI**

CGI is ubiquitous, but it does not scale well. A process is created and destroyed for each client-request exchange. Another problem is that CGI is stateless across client invocations, yet e-Business applications want to maintain shopping cart sessions with many users.

**NSAPI and ISAPI** Earlier, Web server developers began making in-process extensions to their respective servers. These reportedly ran up to a hundred times the throughput of CGI on the same server running on the same computer. These in-process exten-

![Figure D.2  CGI application flow](image-url)
sions originally required the server to be cycled to replace the application. Since they ran as an in-process part of the server, they exposed the server to error and failure. Netscape’s NSAPI and Microsoft’s ISAPI are examples of these APIs.

Later server versions were able to load and unload applications dynamically for version updates. Another improvement was an option to allow an unproven extension to run in a process separate from that of the server. When the extension was deemed reliable it could still be brought into the server process.

**Fast CGI** A variation of CGI retains the out-of-process nature of CGI but doesn’t destroy the process after a stateless request is finished. Instead, it returns the process to a process pool for assignment to another client request. Fast CGI or FGI is faster than CGI, but programmers must be careful to keep the code stateless and not allow a request to access data lying around from the previous use of the process. Users do not like credit card numbers sitting around in some other client’s data fields.

**Java Servlets** Java provides a server application framework called *servlets*. A Java servlet works much like a Java version of CGI except that, once instantiated, it persists in the Java virtual machine. Two variations, internal and external, correspond to in-process and out-of-process non-Java applications, except that a Java virtual machine hosts the servlet within the server process or in a separate process. The servlet is either demand-loaded on first use or is preloaded, depending on a rule set by the server administrator. The Java just-in-time (JIT) compiler compiles the servlet when it is first loaded. The compiled code is re-used for subsequent requests. An advantage of servlets is that the same class binary will execute on different server platforms. The same may be said of the FCGI Perl scripts.

**Session State**

Client-server Web interaction is inherently stateless. That is, the server doesn’t remember what it did for the client during the previous interaction or even whether it has ever seen the client before. This makes it difficult, for instance, to conceive of how to build a customer order over the course of several client-server exchanges.

**Hidden Text Fields** An early solution was to keep state data in a hidden HTML text input field. Figure D.3 shows a simple example of the user’s name being used to customize each subsequent page. This concept could be extended to build a shopping cart.

**Browser Cookies** Browser *cookies* store keyed strings on a per-URL basis. Up to twenty cookies may be stored per URL. They are passed within the HTTP protocol header information. A cookie has an expiration time and date and may be set to persist only until the browser leaves the URL or for weeks.
This is how users customize portal pages today. The server keeps the user information in the user's own browser! Passwords and user IDs for mildly secure sites may also be stored and retrieved in this manner.

A real advantage of cookies is in maintaining state during e-Business or enterprise Web applications. For example a user may visit a site to create an order for the parts to build a computer. A shopping-cart cookie is used to contain catalog numbers and quantities of items selected during several interactions with the server.

Imagine the server view of this operation. The server is getting hit with requests from hundreds of clients. It has pools of threads or processes handling these requests. Now imagine that this server is really a farm of servers. A given client may not have hit a given server earlier in the session. A given thread or process has no idea who the client is or what is in the shopping cart, until it consults the cookie passed with the request. Now it can update the cookie with the current request and send it back to the client with the intermediate output page. The client machine user may think she or he is the only one getting service from the site. (See Figure D.4.)

But wait! The phone rings during the session. The user accepts a luncheon date, returning to the session an hour later. The server responds that the session has timed out and would the user please re-enter the order. This is because the lifetime limit on the cookie expired. The reason for having a expiration is that the "stock" in the shopping cart is logically out of circulation, yet the client hasn’t bought anything. When the cookie expired, the stock was logically returned to the warehouse. The irony here is that clients all over the Web hold or time-out their own session data.

**Figure D.3 User name remembered in hidden field**

1. User fills in initial cgi form, entering her name.
3. Browser sends form and hidden tag to form2app.
4. Output HTML built hidden field’s data for visible name.
Hybrid Session Data

Large amounts of state data may reside in the DBMS data tier. But how do we know where it is? Some sort of session key is needed. A browser cookie or hidden HTML text field may be the key. Higher-level Web APIs have the session concept built in.

Transactions

Web e-Business or intranet enterprise applications need the ACID services of a transaction server. Crafting transactions using traditional transaction servers is somewhat complex. A new paradigm has emerged.

Server-Side Components

Component architecture has made writing transacted Web applications easier. Component models have found their way into Web server-side application logic.

COM / COM+ Microsoft’s model is business logic built on COM components. Microsoft Transaction Server (MTS) shipped for Windows NT 4 in 1996. Logic is coded into one or more COM components. A component may be bound to remote DBMS data such that the data appears to be part of the object instance. Such compo-
nents may be dropped onto a window in MTS Explorer to form a package. This package may be marked with one of several degrees of transaction participation, from “doesn’t use transactions” to “requires transaction.” COM and MTS have come to Windows 2000 as COM+ applications.

COM or COM+ components are usually written to be stateless across transactions. This allows the application to scale upward a great deal. Otherwise, keeping a set of stateful object instances around brings a server to its knees when the goal is to service hundreds of requests. Stateless objects may be pooled or created and destroyed as needed. Each instance is primed with data from the database during execution.

A CGI or ISAPI application may use packages or COM+ applications to form server application logic. A higher-level way is to use them in active server pages.

**Enterprise Java Beans** Some vendors base their middle-tier strategy on Java cross-platform components, a model called enterprise Java beans (EJB). This server-side Java-based component model is cross-platform, naturally. Server farms could consist of servers of different hardware and operating system architectures all running a Java application server using a copy of the same EJB application. There are several different types. The stateless session bean resembles an MTS package or a COM+ application to a remarkable degree. This type of EJB is probably the most scalable.

An enterprise transaction server, part of the EJB runtime, provides the ACID transaction properties for the bean. IBM Websphere Enterprise Edition uses IBM TxSeries for its transaction server.

**Active Server Pages** Web servers have had a capability called server-side include (SSI) almost since the beginning. When SSI is turned on, the server looks at the HTML stream as it sends it to the client instead of simply shipping it. Special tags in the HTML page signal the server to include another file, much as a #include works in the C preprocessor. A variation of the tag says to run a script.

Microsoft generalized this paradigm to active server pages or ASP. (See Code Example D.2.) Thus the server runs scripts in ASP that may emit HTML wrapped around an SQL query result or have other side effects. The two most common scripting languages in ASP are VBScript and JScript, Microsoft’s name for JavaScript or EcmaScript. Other languages may be plugged in, provided they are written to use Microsoft’s COM-based script engine. Thus such script languages must be made COM-enabled. A script may execute MTS or COM+ logic. The page may be marked as requiring a transaction, in which case all COM+ components on the page will participate. What could be easier?

---

If the client user views the HTML source in the browser, he or she will see normal HTML. All of the ASP tags are stripped as the server executes them.

ASP has ancillary benefits. One is that a time-limited session state is built into the design. Several scripting APIs allow setting and getting keyed session data. COM objects are used to automate common Web application tasks such as making dynamic lists of hyperlinks to files.

One advantage of ASP is the remarkable tool set integration available in Microsoft Visual Studio. Almost every part of a transacted Web application is eased and enabled by tools such as Visual Interdev.

A disadvantage of ASP is that it is usually limited to Microsoft’s Internet Information Server (IIS). Third-party vendors market ASP for most other Web servers and platforms, but use of COM objects within those ASPs is a rare phenomenon. This is because COM is inherently Win32-based. MTS or COM+ implementations do not exist on non-Windows platforms, nor are ports expected in the future.

**Java Server Pages**  Java interests countered ASP with server-side includes of their own called Java server pages (JSP). Once you understand ASP, the leap to JSP is simple. The tagged script in the HTML becomes compiled Java (not JavaScript), the component model used by the script becomes EJB, and the transaction server becomes the Enterprise Transaction Server (EJS) used in the EJB runtime.

The W3C open source Web server platform is Jigsaw, a cross-platform Java-based application server. It will accept any of several JSP implementations, such as

```html
<html>
<head><title>ASP Snippet</title></head>
<body onload="document.cookie='cart_items='+0' ">
<%session.session.timeout = 40%>
<p>Today is <%=WeekdayName(Weekday(Date))%><br>
<% If Request.Cookies("Cust").HasKeys Then
  cname = Request.Cookies("Cust")("Customer")
  email = Request.Cookies("Cust")("Email")
  End If
%>
...
</body>
</html>

**Code Example D.2**  Active server page snippet
GNUJSP, RocketJSP, zJsp, Caucho’s Resin, or JSWDK from Sun. IBM’s WebSphere Enterprise Edition is a complete JSP application server with support for enterprise Java beans.

An advantage of JSP is that it will operate on a variety of hardware and software server platforms. Its Java-based applications may also communicate with legacy platforms using IIOP COBRA techniques.

Security

A distributed application must authorize and authenticate. Identification and authentication is the process of determining who the communicating parties are. Authorization is about what those parties are allowed to access.

Basic Web authentication simply has the client send a clear-text password over the public Internet. This is a simple tool to force users to give demographic data once to get the login ID and password to proceed into a site. Basic Web authentication is not useful to protect sites or applications from unauthorized users.

Many Web servers support higher forms of authentication than are covered in this book. The Windows NT LAN Manager (NTLM) challenge-response mechanism used in Windows NT is common, but it doesn’t support delegation of trust to succeeding parties. Other schemes are also used, such as shared secret authentication through a trusted third party (Kerberos) and X.500 certificate authentication.

Authorization is handled by the operating system that hosts the Web application. The security token obtained during authentication may be compared to an access control list associated with a file or directory to allow or deny the end user access to the information.

15. See http://www.w3.org/Jigsaw/.
As Internet technologies proliferate, the need for applications to implement security above transmission control protocol (TCP) increases. GSS-API and SSPI are two similar interfaces for accessing distributed authentication protocols. The GSS-API is standardized in Internet Engineering Task Force RFCs 2743 (Generic Security Service Application Program Interface: Version 2, Update 1) and 2744 (Generic Security Service API Version 2; C-bindings). Microsoft currently documents the SSPI in the Windows Platform SDK under Security, Logon Authentication, Security Support Provider Interface. They are interoperable when using the Kerberos authentication mechanism. This appendix provides a summary of the APIs and a sample showing how to use each.

The individual functions in both APIs can be divided into the following categories.1

1. General helper functions—These functions provide buffer management, name format translation, and other utility services required for using the interface.
2. Credential management functions—These provide the interfaces for clients and servers to manage their own authentication credentials.
3. Context management functions—These provide the interfaces to acquire and process security tokens that are used to establish a common, secured context between the client and server.
4. Message protection functions—These provide the interfaces with a security context to protect individual messages from either disclosure or modification.

---

1. Please note that neither RFC 2743 GSS-API Version 2 nor Microsoft’s SSPI documentation classify the functions into these four categories.
The main differences among the interfaces are in name and buffer management. The SSPI uses Windows user IDs while the GSS-API supports a multitude of mechanism-specific name types. Also, the SSPI uses strings to name security mechanisms. In the GSS-API, security mechanisms are named using unique OIDs. The interfaces also tend to do buffer management differently. Where one interface will require separate input and output buffers, the other will use a single buffer, overwriting the input with the output.

**General Helper Functions**

**Mechanism/Package Management**

Both interfaces include functions that are used either to initialize access to the interface or to retrieve information about the security mechanisms supported by the system. `EnumerateSecurityPackages` with `QuerySecurityPackageInfo` or `gss_indicate_mechs` can be used to find a security mechanism that supports application requirements. (See Table E.1.)

The programs `sspiqsp` and `sspipkg` on the CD demonstrate the use of the `EnumerateSecurityPackages` and `QuerySecurityPackageInfo` SSPI functions.

<table>
<thead>
<tr>
<th>Function Purpose</th>
<th>GSS-API</th>
<th>SSPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads the DLL that supports the SSPI.</td>
<td></td>
<td><code>InitSecurityInterface</code></td>
</tr>
<tr>
<td>Gets information about the security mechanisms that are supported.</td>
<td><code>gss_indicate_mechs</code></td>
<td><code>EnumerateSecurityPackages</code></td>
</tr>
<tr>
<td>Gets information about a single security mechanism.</td>
<td></td>
<td><code>QuerySecurityPackageInfo</code></td>
</tr>
</tbody>
</table>

**Buffer Management**

Both interfaces allocate memory for buffers that are returned to the using application. Because of this both provide functions that must be used to release these buffers. (See Table E.2.)
Name Management Functions

The GSS-API allows each security mechanism to continue to use its mechanism specific user naming. This requires the interface to allow names of different formats to be tagged and imported. (See Table E.3.)

Object Identifier Helper Functions

The GSS-API makes extensive use of ISO-based object identifiers used to identify security mechanisms as well as the name space of user names. The OID helper functions assist programmers in working with the OIDs. (See Table E.4.)
PART 5 ▪ APPENDICES, REFERENCES, AND GLOSSARY

Other Functions

Additionally the GSS-API has a function to convert its numeric status codes into printable messages. For the SSPI, this function is done using the Win32 functions GetLastError and FormatMessage. (See Table E.5.)

<table>
<thead>
<tr>
<th>Function Purpose</th>
<th>GSS-API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Converts an OID in string into its encoded form.</td>
<td>gss_str_to_oid</td>
</tr>
<tr>
<td>Converts an encoded OID into a printable string.</td>
<td>gss_oid_to_str</td>
</tr>
<tr>
<td>Frees the storage associated with an OID.</td>
<td>gss_release_oid</td>
</tr>
<tr>
<td>Creates an empty set of OIDs.</td>
<td>gss_create_empty_oid_set</td>
</tr>
<tr>
<td>Adds an OID to an OID set.</td>
<td>gss_add_oid_set_member</td>
</tr>
<tr>
<td>Checks to see whether an OID is already in an OID set.</td>
<td>gss_test_oid_set_member</td>
</tr>
<tr>
<td>Frees an OID set created with gss_create_empty_oid_set.</td>
<td>gss_release_oid_set</td>
</tr>
</tbody>
</table>

Table E.5 Other functions

<table>
<thead>
<tr>
<th>Function Purpose</th>
<th>GSS-API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translates numeric status codes into printable strings.</td>
<td>gss_display_status</td>
</tr>
</tbody>
</table>

Credential Management Functions

The credential management functions are used to get access to the logon credentials for a security principal. Credentials, such as user IDs and passwords, might be stored in a Kerberos keytab file, a Kerberos ticket-granting ticket, or other information that is used to prove an identity in an authentication exchange. (See Table E.6.)
Context Management Functions

The context management functions are used to create and use a common security context between two programs. Each context contains information specific to a single set of authenticated exchanges. For example, the context may contain a session key used for the encryption and decryption of messages.

The process of creating a context is basically the same for both interfaces. The client issues initialize\(^2\) in a loop until it receives a complete status. On each pass it forwards the returned security token to the companion server. The server performs an equivalent loop using accept, with the token from the client as input, and sends back any output token. In pseudocode the client loop is as follows.

\[
\text{SecContext} = \text{NULL}; \\
\text{Do} \\
\quad \text{Rc} = \text{Initialize(SecContext, InputToken, OutputToken)}; \\
\quad \text{Send OutputToken}; \\
\quad \text{If (rcl=continue\_needed AND rcl=complete)) Then Exit Failure;} \\
\quad \text{If (rc=continue\_needed) Then wait to receive InputToken from server;} \\
\quad \text{While (rcl=complete)}
\]

---

\(^2\) InitializeSecurityContext for the SSPI, gss_init_sec_context for the GSS-API.
The programs in the sspi_sample and gss_sample subdirectories on the CD demonstrate the main loop processing for both interfaces. The samples interoperate with each other, demonstrating how programs written for GSS-API can communicate with programs written using SSPI. The communications for gss_sample are illustrated in Code Example E.1.

Code Example E.1 outlines the program logic for using the GSS-API in the sample programs on the CD. The client calls `gss_init_sec_context()`—generating tokens that are sent to the server program. The server program uses `gss_accept_sec_context()` to process the received tokens. This can result in the creation of additional tokens. Both programs continue looping as long as the GSS-API indicates `Continue_Needed`. Once authentication is completed, the client encrypts a message using `gss_wrap` and sends it to the server. The server decrypts the message with `gss_unwrap` and computes a message integrity code (using `gss_get_mic`), which is sent back to the client. The client verifies the integrity code with `gss_verify_mic`. Both client and server use `gss_delete_sec_context` to clean up the security context after they are done. The same programming logic works with the SSPI, the difference being the names of the functions that are called. (See Table E.7.)

```plaintext
gss_client
do {
  gss_init_sec_context()
  if token_to_send
    send token to server
  if ContinueNeeded
    receive token from server
} while ContinueNeeded

Send encrypted message
Receive message integrity code
Receive encrypted message
Send message integrity code

gss_server
do {
  receive token from client
  gss_accept_sec_context()
  if token_to_send
    send token to client
} while ContinueNeeded

Receive encrypted message
Send message integrity code

Code Example E.1  gss_sample client-server exchange
### Table E.7  Context management

<table>
<thead>
<tr>
<th>Function Purpose</th>
<th>GSS-API</th>
<th>SSPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used by a client to create a security context, retrieve the tokens to send to the server, and process tokens received from the server</td>
<td>gss_init_sec_context</td>
<td>InitializeSecurityContext</td>
</tr>
<tr>
<td>Used by a server to create a security context, process tokens received from the client, and retrieve tokens to send to the client</td>
<td>gss_accept_sec_context</td>
<td>AcceptSecurityContext</td>
</tr>
<tr>
<td>Used to destroy a security context created by either initialize or accept</td>
<td>gss_delete_sec_context</td>
<td>DeleteSecurityContext</td>
</tr>
<tr>
<td>Used to process a token sent by the partner when they have deleted their security context (GSS-API Version 1 is no longer recommended.)</td>
<td>gss_process_context_token</td>
<td></td>
</tr>
<tr>
<td>Retrieves information about the security context</td>
<td>gss_inquire_context</td>
<td>QueryContextAttributes</td>
</tr>
<tr>
<td>Creates a Windows access token that can be used in Windows security calls</td>
<td>gss_wrap_size_limit</td>
<td>QuerySecurityContextToken</td>
</tr>
<tr>
<td>Sets the Windows 2000 security environment (i.e., access token) for a thread to the received security context</td>
<td>gss_context_time</td>
<td>ImpersonateSecurityContext</td>
</tr>
<tr>
<td>Restores a server thread's security environment (i.e., access token) to its state before the usage of ImpersonateSecurityContext</td>
<td>gss_export_sec_context</td>
<td>RevertSecurityContext</td>
</tr>
<tr>
<td>Exports the security context into a buffer that can be used to recreate it in another process</td>
<td>gss_import_sec_context</td>
<td>ExportSecurityContext</td>
</tr>
<tr>
<td>Creates a context from a previously exported buffer</td>
<td>gss_wrap_size_limit</td>
<td>ImportSecurityContext</td>
</tr>
<tr>
<td>Used to apply special control tokens to existing security contexts</td>
<td>gss_context_time</td>
<td>ApplyControlToken</td>
</tr>
<tr>
<td>Used in cases where the entire token is not provided on the call to initialize</td>
<td>gss_inquire_context</td>
<td>CompleteAuthToken</td>
</tr>
</tbody>
</table>
Message Protection Functions

The message protection functions are used to create modified messages that are encrypted (for confidentiality protection) or signed (for integrity protection). Both interfaces require that a security context already be established. The gss_sample and sspi_sample programs include these functions to both encrypt a message and to verify a message. (See Table E.8.)

<table>
<thead>
<tr>
<th>Function Purpose</th>
<th>GSS-API</th>
<th>SSPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encrypts a message to protect it from disclosure.</td>
<td>gss_wrap</td>
<td>EncryptMessage</td>
</tr>
<tr>
<td>Decrypts a received message.</td>
<td>gss_unwrap</td>
<td>DecryptMessage</td>
</tr>
<tr>
<td>Creates a message signature that can be used to detect whether a message has been modified.</td>
<td>gss_get_mic</td>
<td>MakeSignature</td>
</tr>
<tr>
<td>Verify a message signature to determine whether a message has been modified.</td>
<td>gss_verify_mic</td>
<td>VerifySignature</td>
</tr>
</tbody>
</table>
A critical component of effective security is controlling who is allowed to view or modify data (or execute a program). Storing the data within a directory does not remove this need. Access control standards for LDAP accessible directories do not yet exist. While the LDAP extensions workgroup of the IETF is working on this area, a consensus on a single access control model for directories does not exist. This appendix describes the access control model for three major directory services: the IBM SecureWay Directory access control model submitted to the IETF, the Microsoft Active Directory model (based on the Windows NT model), and Netscape Directory Server. Dealing with the differences among these models is one of the challenges in writing directory neutral products.

IBM SecureWay Directory Access Control

The IBM SecureWay Directory access control model was submitted to the IETF as Internet Draft draft-ietf-ldapext-acl-model-04.txt. This document is included on the CD; see Appendix H for a reference and link.

Users and Groups in Access Control Lists

The IBM SecureWay Directory controls access based on users, groups, and roles. Users normally have the object class of person or eUser. Security groups must be of class AccessGroup. In addition, security roles must be of class AccessRole. Access groups and access roles are types of groups. Since each user, group, and role is a directory object and has a unique distinguished name (DN), they do not have to be at predetermined locations in the directory information tree. The DN tagged with a type is used in entryOwner and aclEntry attributes. The tags are access-id, group, and role. For example, an entryOwner might be access-id: CN=Rich, O=IBM, C=US, role: CN=Developer, OU=SWG, O=IBM, C=US, or role: CN=author, O=AWL, C=US. These are also several pseudo-DNs that can be used to represent specific categories of users, as shown in Table F.1.
The access control list information for a directory object is stored in six attributes, shown in Table F.2.

Access entries consist of an access class followed by a set of permissions. The classes are object, normal, sensitive, and critical. The object access class refers to the object itself. Useful in giving users the ability to modify the directory information about them.

### Structure of an Access Control List

The access control list information for a directory object is stored in six attributes, shown in Table F.2.

Access entries consist of an access class followed by a set of permissions. The classes are object, normal, sensitive, and critical. The object access class refers to the object itself. Useful in giving users the ability to modify the directory information about them.
as a whole and may have the permissions of “a” (add) and “d” (delete). The other classes refer to types of attributes and may have the permissions of “r” (read), “w” (write), “s” (search), and “c” (compare).

Each attribute defined by the schema is assigned to a single access class in the attribute's schema definition. This makes it critical for attributes to be grouped according to the applicable security policy when the schema is created.

**Evaluation Algorithm for an Access Check**

The evaluation algorithm is as follows.

If user's bind DN has a matching aclEntry then
  Grant only the permissions explicitly given to the DN.
Else if Any of the user's groups have matching aclEntry values then
  Grant the union of all permissions granted to the groups.
  Also include the permissions for the cn=Anybody group.
Else
  Grant only the permissions assigned to the cn=Anybody group.
End If

One way to think of this algorithm is that the user's more explicit permissions completely replace any permission that he or she might have based on group or role memberships.

**Creating a New Object**

Access control list propagation is handled during the access check evaluation. Given this, no special processing is required to be performed at object or attribute creation time.

**Common Problems and Solutions**

**Granting Anonymous Access** Granting anonymous access is easy. It consists of adding a single aclEntry granting the desired permissions to the Anybody group. For example

```
group:cn=Anybody:normal:rsc
```

grants anonymous read, search, and compare but does not grant write access.

**Allowing Modification** To allow a user to modify some of his or her own attributes, add an aclEntry granting the desired permissions to the access ID:

```
Access-id:cn=this:normal:rwsc:critical:sc
```
Preventing Access  Because it currently lacks an explicit deny ability, preventing an individual from accessing a specific directory is not always straightforward. One technique that can work in many cases is to explicitly grant the user a very small set of permissions (maybe compare only). This will block him or her from using permissions associated with group memberships.

Microsoft Active Directory Access Control

Users and Groups in Access Control Lists

Security principals in Microsoft Active Directory are represented in several ways. First users are objects of class user, including the classes that derive from it, like computer. Groups are objects of class group. If the group is meant to be used for access control, the attribute groupType will be set, indicating what type of security group it is. Because they are directory objects, users and groups have distinguished names and have additional unique representations. Table F.3 lists some of the user and group attributes that provide the additional representation.

Ultimately, the important values are the security identifiers (SIDs) related to the users and groups. SIDs are assigned using an algorithm and include a representation of the domain, which means that SIDs are unique.

Structure of an Access Control List

The Windows 2000 access control data is stored in the ntSecurityDescriptor attribute of an object. ntSecurityDescriptor is a mandatory attribute of the top class. Because of this, every object in the Active Directory will have a value for the ntSecurityDescriptor.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>samAccountName</td>
<td>The short name used for down-level communications. It must be unique across a domain.</td>
</tr>
<tr>
<td>userPrincipalName</td>
<td>A friendly name of the form samAccountName@domain. It can be administratively managed, so it could be company e-mail addresses, for example. Must be unique within the forest.</td>
</tr>
<tr>
<td>SidHistory</td>
<td>A list of “old” security identifiers that are still related to the user. This can include user and group SIDs from domains that no longer exist.</td>
</tr>
<tr>
<td>securityIdentifier</td>
<td>A unique byte string (it does contain internal structure) that identifies the user or group. This value is generated such that it should be unique forever.</td>
</tr>
</tbody>
</table>
attribute. The structure of a security descriptor is shown in Figure F.1. Security descriptors are stored in binary form. Fundamentally, this restricts the manipulation of ntSecurityDescriptors to Windows systems. The Active Directory Services Interface (ADSI) includes methods specific to manipulating security descriptors.

The owner and owning group fields control some special rights to an object. For example, the owner is always allowed to modify the security descriptor. The discretionary access-control list (DACL) contains the access control list (ACL) that is used in making access decisions. The system access-control list (SACL) also has an ACL, but it is used to control when audit records are recorded. Both ACLs consist of an ordered list of access control entries (ACE). There are two main types of ACEs, regular and object. The regular ACEs are unchanged from Windows NT. The object ACEs include up to two GUIDs. One GUID identifies the attribute type or object

![Security descriptor diagram](image)

**Figure F.1** Structure of ntSecurityDescriptor
class the entry applies to. The other GUID restricts the classes of child objects that will inherit the ACE entry.

**Evaluation Algorithm for an Access Check**

The evaluation check algorithm is used to scan the DACL ACE entries in order. An ACE applies when the subject SID matches one of the SIDs in the user’s access token. The access token contains the user’s SID, all of the SIDs for the groups he or she is a member of, and all SIDs from the sidHistory attribute. The next check is to see whether the ACE applies to the attribute or object being accessed. This check is done by comparing the GUID in the ACE with the schema GUID for the attribute or class. Assuming that the GUIDs match, if the ACE is a deny ACE and any of the permission bits in the ACE mask are being requested, the request fails. If the GUIDs match and the ACE is an allow ACE, any matching permissions are added to a set of accumulated permissions and if all request permissions have been found, the request is granted. If the entire DACL is scanned without accumulating all request permissions, the request fails.

**Creating a New Object**

ACL inheritance is dynamic, in that the addition, change, or deletion of an inheritable ACE is reflected to the objects below it in the directory tree. This is done by recomputing the ACLs of all subobjects. When an object is created, its initial ACL is set using the inheritable ACE entries of its immediate parent.

**Common Problems and Solutions**

**Granting Anonymous Access** Allowing anonymous access consists of modifying the directory ntSecurityDescriptors to allow the “guest” account permissions to the object and its ancestors. Anonymous access to an entire subtree can be done using ACE inheritance. By default, Active Directory only allows anonymous access to the top-most object in a domain.

**Allowing Modification** To allow a user to modify his or her own data, an allow ACE is added with the user’s SID as the subject and the permissions are set to allow read, write, and so on. The ACE is set to the schemaGUID for the applicable property. If update is to be granted to multiple attributes, then multiple ACEs will need to be added.

**Preventing Access** To prevent an individual from accessing an object’s attribute add a deny ACE to the beginning of the DACL. The deny needs to apply to the user’s
SIDs and the GUID of the attribute. Once again, to deny access to multiple
attributes, it is likely that multiple ACEs will be needed.

Netscape Directory Access Control

Users and Groups in Access Control Lists

Netscape does not enforce a strict notion of user objects. Any directory object that
has the userPassword attribute can be a user. Most of the time this applies to objects
of the class user, but other object classes can also include the attribute.

Groups are objects of class groupOfNames or groupOfUniqueNames. Both of these
object classes maintain a list of members.

Structure of an Access Control List

Each directory object may contain one or more aci entries. The aci attribute is an
optional attribute of the top class. Each aci entry consists of three pieces of informa-
tion: target, permissions, and bind rules. The target defines which objects and
attributes the entry refers to. Permissions determine the operations that the entry
controls. Bind rules are used to decide whether the entry applies to how the LDAP
client application that is making the request was bound to the directory. This is based
on the information provided with the ldap_bind operation.

Target information Target information determines the exact objects and attributes
the aci applies to. Target information is used to control the inheritance of aci informa-
tion by objects lower in the directory tree as well as restricting an aci to apply to spe-
cific attributes. Targets come in two forms. The first, a DN form, does support the use
of wild-card characters in the most-specific RDN portion. For example, a target of
CN=R*,OU=SWG,O=IBM,C=US applies to all entries that start with the letter R in the
OU=SWG,O=IBM,C=US directory container. The other form for a target is an ldap search
filter. The filter form allows the specification of individual attributes to protect.

Permissions The Netscape Directory Server supports the access permissions in
Table F.4.

Bind Rules Similar to the way the target determines that the aci applies to the object
or attribute, bind rules are used to decide whether the aci is to be applied to the cur-
rent ldap request. A bind rule includes an expression that is matched to the current
connection environment. The matching can be based on many items, including user
and group DNs from the ldap_bind, the IP address of the client system, day or
time of the request, and the type of authentication that was performed. A bind rule can be set to apply if the match is true or if the match is false. This means that a bind rule might apply for all user DN except one. The actual form is one of the following.

- `<keyword> = "<expression>";`
- `<keyword> != "<expression>";`

The equals sign (=) means that the aci applies if the bind rule matches; while the exclamation-equals (!=) sign means the aci applies if the expression is not matched. The keyword identifies what is being looked for in the current environment. Some keywords are userdn, groupdn, ip, and dayofweek. There are other supported keywords. For userdn, two special expression values are `ldap://anyone` and `ldap://self`.

### Composed ACI

When composed an aci has the form

```plaintext
aci: (<target>)(version 3.0;aci"<name">;<permission><bind rule>;)
```

Target, permissions, and bind rule have already been discussed. Name allows the administrator to describe the aci and is required. “version 3.0” is a version identifier and is also required.

### Evaluation Algorithm for an Access Check

The access check algorithm is to
- collect all ACIs that target the requested attributes or object.
- collect the permissions from the ones whose binding rules apply. (Remember the binding rule might apply because the match fails, e.g., != was used as the operator.)
- if any of the requested permissions are denied, fail the request; otherwise allow the request only if all needed permissions were allowed.

The collection set uses the aci value from the object and all of its ancestors.

**Common Problems and Solutions**

**Granting Anonymous Access**  Anonymous access is granted by creating an aci that targets the interesting objects/attributes and using a binding rule to allow access by ldap://anyone. For example

```
(target="ldap://O=IBM") (targetattr=*)
(version 3.0; aci "anonymous to IBM"; allow (read, search, write)
userdn = "ldap://anyone");
```

will grant read, search, and write privilege to an anonymous user to the entire O=IBM directory tree.

**Allowing Modification**  The following aci will allow a user to modify all attributes of the object he or she binds to the directory as

```
(targetattr = *)
(version 3.0; aci "change of all users attributes"; allow (write)
userdn = "ldap://self");
```

**Preventing Access**  The following aci will prevent a user from accessing a specific DN.

```
(target="ldap://protected dn")
(version 3.0; aci "keep joe away"; deny (all)
userdn = "ldap://joe dn");
```
The IBM standard schema incorporates many industry standard features. The definition is large, containing hundreds of objects. An HTML reference of the entire standardized schema is on the CD.

Standard Directory Information Tree (DIT)

Figure G.1 illustrates the top of IBM's standard directory information tree.

Standard Objects

All object classes derive from object class top, hierarchy level one. Some level-two object classes are

- accessGroup
- accessRole
- account
- alias
- aliasObject
- applicationEntry
- bootableDevice
- certificationAuthority
- changeLogEntry
- container
- domain
- locality

There are many more level-two entries. Please see the HTML rendering of the schema on the CD.
Attributes

IBM has a huge number of attributes in its schema. Usually, an attribute’s purpose is clearly self-described by its name. A small set of examples is

- authenticationHost
- certifierPassword
- countryCode
- creatorsName (No, IBM is not into theology!)

Syntaxes

Here are a few examples of syntaxes in the IBM standard schema, along with some clearly named client attributes of each syntax.

Binary  This syntax is used for bits and BLOB attributes. Some attributes having binary syntax are

- audio
- javaSerializedObject
- secretKey
- userPassword

**Boolean** Yes/no attributes use the boolean syntax. Some attribute examples are

- audibleAlarm
- compressed
- editable
- isAccountEnabled

**GeneralizedTime** Date and time attributes use the GeneralizedTime syntax. Examples of attributes using this syntax are

-actionDate
- lastModifiedTime
- passwordExpireTime
- timeOfLastReset

**TelephoneNumber** The telephoneNumber syntax is used by attributes that are telephone numbers. Some examples follow.

- cellular TéléphoneNumber
- homeFax
- pager
- telephoneNumber
Appendix H

DIRECTORY STANDARDS AND OTHER REFERENCES

Note that on the companion CD, the Appendix H web and FTP links are live.

IETF RFCs

This section lists the major directory and security-related Internet RFCs. A complete list of all Internet RFCs can be accessed at http://www.ietf.org/rfc.html.

RFC 883: Domain Names—Implementation and Specification

CD-ROM: \Appendx\App_H\rfcs\RFC883.txt

Description: RFC 883 documents the definition of domain name servers and resolvers, the functions they perform, and the protocols they use to communicate. RFC 1035 obsoletes this RFC.

RFC 1035: Domain Names—Implementation and Specification

CD-ROM: \Appendx\App_H\rfcs\RFC1035.txt

Description: RFC 1035 documents the definition of domain name servers and resolvers, the functions they perform, and the protocols they use to communicate. RFC 2535 obsoletes this RFC.
RFC 1508: Generic Security Service Application Program Interface

**URL:** ftp://ftp.isi.edu/in-notes/rfc1508.txt

**CD-ROM:** \Appendx\App_H\rfcs\RFC1508.txt

**Description:** RFC 1508 documents the functions provided by the Generic Security Service Application Program Interface (GSS-API). This API enables mechanism-independent authentication, data confidentiality, and data integrity services. RFC 2078, documenting version 2 of the GSS-API, obsoletes this RFC.

RFC 1509: Generic Security Service API: C-bindings

**URL:** ftp://ftp.isi.edu/in-notes/rfc1509.txt

**CD-ROM:** \Appendx\App_H\rfcs\RFC1509.txt

**Description:** The C language bindings that match RFC 1508. This has the C-specific definitions required for using the GSS-API. Obsoleted by RFC 2744 for GSS-API Version 2.

RFC 1510: The Kerberos Network Authentication Service (V5)

**URL:** ftp://ftp.isi.edu/in-notes/rfc1510.txt

**CD-ROM:** \Appendx\App_H\rfcs\RFC1510.txt

**Description:** The Kerberos third-party authentication service was created at MIT. RFC 1510 describes version 5 of Kerberos. It includes the protocol flows and formats between the client, server, and authentication service. An Internet draft, draft-ietf-cat-kerberos-revisions-04.txt, is currently being worked on to replace this RFC. The draft clarifies many areas of RFC 1510.

RFC 1750: Randomness Recommendations for Security

**URL:** ftp://ftp.isi.edu/in-notes/rfc1750.txt

**CD-ROM:** \Appendx\App_H\rfcs\RFC1750.txt

**Description:** Recommendations on how to create random numbers when needed by security mechanisms.


**URL:** ftp://ftp.isi.edu/in-notes/rfc1777.txt

**CD-ROM:** \Appendx\App_H\rfcs\RFC1777.txt
Description: RFC 1777 documents the early definition of the LDAP protocol. RFC 2251 obsoletes this RFC.

RFC 1823: The LDAP Application Program Interface

CD-ROM: \Appendx\App_H\rfcs\RFC1823.txt

Description: Information RFC that documents the LDAP API. This RFC is the starting place for writing applications that use LDAP.

RFC 1964: The Kerberos Version 5 GSS-API Mechanism

CD-ROM: \Appendx\App_H\rfcs\RFC1964.txt

Description: Describes the format of the Kerberos tokens when using the GSS-API. It includes other information specific to using Kerberos and the GSS-API together.

RFC 2078: Generic Security Service Application Program Interface, Version 2

CD-ROM: \Appendx\App_H\rfcs\RFC2078.txt

Description: The second version of the GSS-API. This version has been made obsolete by RFC 2743.

RFC 2222: Simple Authentication and Security Layer (SASL)

URL: ftp://ftp.isi.edu/in-notes/rfc2222.txt
CD-ROM: \Appendx\App_H\rfcs\RFC2222.txt

Description: SASL is a layer that adds user authentication and security. It is designed for connection-oriented protocols. SASL is used by LDAP for security.

RFC 2245: Anonymous SASL Mechanism

CD-ROM: \Appendx\App_H\rfcs\RFC2245.txt

Description: An SASL mechanism designed to allow anonymous access.
RFC 2251: Lightweight Directory Access Protocol (v3)

**URL:** ftp://ftp.isi.edu/in-notes/rfc2251.txt

**CD-ROM:** \Appendx\App_H\rfcs\RFC2251.txt

**Description:** The RFC that describes the functions provided by LDAP. This includes the types of operations that are supported, along with the wire formats and protocols.


**URL:** ftp://ftp.isi.edu/in-notes/rfc2252.txt

**CD-ROM:** \Appendx\App_H\rfcs\RFC2252.txt

**Description:** This RFC defines the attribute syntaxes that should be implemented by an LDAP-based directory. These are currently recommendations and not requirements.


**URL:** ftp://ftp.isi.edu/in-notes/rfc2253.txt

**CD-ROM:** \Appendx\App_H\rfcs\RFC2253.txt

**Description:** Documents how distinguished names are formatted when sent between the directory client and directory server.

RFC 2254: The String Representation of LDAP Search Filters

**URL:** ftp://ftp.isi.edu/in-notes/rfc2254.txt

**CD-ROM:** \Appendx\App_H\rfcs\RFC2254.txt

**Description:** Documents how search filters are formed and transmitted.

RFC 2255: The LDAP URL Format

**URL:** ftp://ftp.isi.edu/in-notes/rfc2255.txt

**CD-ROM:** \Appendx\App_H\rfcs\RFC2255.txt

**Description:** Documents how URLs are formed for LDAP operations.
RFC 2256: A Summary of the X.500(96) User Schema for Use with LDAPv3

**URL:** ftp://ftp.isi.edu/in-notes/rfc2256.txt
**CD-ROM:** \Appendx\App_H\rfcs\RFC2256.txt

**Description:** Translates part of the X.500 schema into an LDAP schema.

RFC 2478: The Simple and Protected GSS-API Negotiation Mechanism

**URL:** ftp://ftp.isi.edu/in-notes/rfc2478.txt
**CD-ROM:** \Appendx\App_H\rfcs\RFC2478.txt

**Description:** Defines a GSS-API authentication mechanism that can be used to negotiate the actual authentication mechanism to be used.

RFC 2535: Domain Name System Security Extensions

**URL:** ftp://ftp.isi.edu/in-notes/rfc2535.txt
**CD-ROM:** \Appendx\App_H\rfcs\RFC2535.txt

**Description:** RFC 2535 documents the definition of domain name servers and resolvers, the functions they perform, and the protocols they use to communicate.


**URL:** ftp://ftp.isi.edu/in-notes/rfc2589.txt
**CD-ROM:** \Appendx\App_H\rfcs\RFC2589.txt

**Description:** Documents LDAP extensions to support the usage of data that has a short lifetime.

RFC 2712: Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)

**URL:** ftp://ftp.isi.edu/in-notes/rfc2712.txt
**CD-ROM:** \Appendx\App_H\rfcs\RFC2712.txt

**Description:** Documents the changes to TLS (the follow-on for SSL) that are required to use Kerberos for authentication.
RFC 2743: Generic Security Service Application Program Interface: Version 2, Update 1

**URL:** ftp://ftp.isi.edu/in-notes/rfc2743.txt
**CD-ROM:** \Appendx\App\rfcs\RFC2743.txt

**Description:** The latest version of the GSS-API.

RFC 2744: Generic Security Service API Version 2: C-bindings

**URL:** ftp://ftp.isi.edu/in-notes/rfc2744.txt
**CD-ROM:** \Appendx\App\rfcs\RFC2744.txt

**Description:** Documents the C language interfaces for the GSS-API. Updates and obsoletes the interfaces documented in RFC 1509.

### IETF Drafts

Internet drafts are not permanent documents. This section offers snapshots of important drafts. They are replaced on an ongoing basis. It is very likely that some of these documents will no longer be available. Others will have been replaced with more recent drafts, and some will have become new Internet RFCs. The starting place for locating current information about Internet drafts is http://www.ietf.org/ID.html.

**Internet Draft: Microsoft LDAP Control for Directory Synchronization**

**URL:** http://www.ietf.org/internet-drafts/draft-armijo-ldap-dirsync-01.txt
**CD-ROM:** \Appendx\App\drafts\draft-armijo-ldap-dirsync-01.txt

**Description:** The Microsoft LDAP dirsync control. This control allows a program to poll for changes to the Active Directory.

**Internet Draft: The LDAP Data Interchange Format (LDIF) - Technical Specification**

**URL:** http://www.ietf.org/internet-drafts/draft-good-ldap-ldif-05.txt
**CD-ROM:** \Appendx\App\drafts\draft-good-ldap-ldif-05.txt

**Description:** The current proposal for a data interchange format for LDAP. LDIF is normally used to do bulk load and unload of directory data. Because the
LDIF format is not completely standardized yet, different directories support various versions of LDIF.

Internet Draft: Generic Security Service API Version 2: Java Bindings


*CD-ROM*: \Appendx\App_H\drafts\draft-ietf-cat-gssv2-javabind-05.txt

*Description*: The Java language information required to use GSS-API Version 2. Java binding was not created for GSS-API version 1.

Internet Draft: The Kerberos Network Authentication Service (V5)


*CD-ROM*: \Appendx\App_H\drafts\draft-ietf-cat-kerberos-revisions-04.txt

*Description*: An updated version of the RFC that documents Kerberos. This draft is intended to replace RFC 1510.

Internet Draft: SASL GSS-API Mechanisms


*CD-ROM*: \Appendx\App_H\drafts\draft-ietf-cat-sasl-gssapi-00.txt

*Description*: Documents how GSS-API mechanisms can be used under SASL.

Internet Draft: Access Control Model for LDAP


*CD-ROM*: \Appendx\App_H\drafts\draft-ietf-ldapext-acl-model-04.txt

*Description*: A proposed access control model for LDAP directories. It is unlikely that the IETF will choose a single access control model anytime soon.

Internet Draft: Access Control Requirements for LDAP


*CD-ROM*: \Appendx\App_H\drafts\draft-ietf-ldapext-acl-reqts-03.txt

*Description*: A set of requirements for access control in LDAP directories.
Internet Draft: The C LDAP Application Program Interface

**CD-ROM:** \Appendx\App\H\drafts\draft-ietf-ldapext-ldap-c-api-04.txt

**Description:** The current draft of the C language binding for version 3 of LDAP.

Internet Draft: The Java LDAP Application Program Interface

**URL:** http://www.ietf.org/internet-drafts/draft-ietf-ldapext-ldap-java-api-09.txt
**CD-ROM:** \Appendx\App\H\drafts\draft-ietf-ldapext-ldap-java-api-09.txt

**Description:** The current draft of Java bindings for version 3 of LDAP.

Internet Draft: Definition of the inetOrgPerson LDAP Object Class

**URL:** http://www.ietf.org/internet-drafts/draft-smith-ldap-inetorgperson-04.txt
**CD-ROM:** \Appendx\App\H\drafts\draft-smith-ldap-inetorgperson-04.txt

**Description:** Defines the inetOrgPerson class.

Internet Draft: The Java SASL Application Program Interface

**URL:** http://www.ietf.org/internet-drafts/draft-weltman-java-sasl-02.txt
**CD-ROM:** \Appendx\App\H\drafts\draft-weltman-java-sasl-02.txt

**Description:** Provides Java class definitions required to access SASL from Java applications.

Internet Draft: Public Key Cryptography for Initial Authentication in Kerberos

**CD-ROM:** \Appendx\App\H\drafts\draft-ietf-cat-kerberos-pk-init-10.txt

**Description:** Adds a public key-based front end to Kerberos authentication. This allows the usage of smart cards and certificates instead of user memorized passwords.

Internet Draft: Extending Change Password for Setting Kerberos Passwords

**URL:** http://www.ietf.org/internet-drafts/draft-ietf-cat-kerberos-set-passwd-00.txt
**CD-ROM:** \Appendx\App\H\drafts\draft-ietf-cat-kerberos-set-passwd-00.txt
**Description:** Extends the Kerberos change password to allow a user’s password to be set by an administrator.

*Internet Draft: Internet Security Glossary*

- **URL:** http://www.ietf.org/internet-drafts/draft-shirey-security-glossary-02.txt
- **CD-ROM:** \Appendx\App_H\drafts\draft-shirey-security-glossary-02.txt

**Description:** A glossary of security-related terms.

**Other IETF Documents**

*Internet Best Current Practice 21: Expectations for Computer Security Incident Responses*

- **URL:** ftp://ftp.isi.edu/in-notes/bcp/bcp21.txt
- **CD-ROM:** \Appendx\App_H\others\bcp21.txt

**Description:** Describes the functions that an incident-response team needs to be able to perform.

*Internet FYI 8: Site Security Handbook*

- **URL:** ftp://ftp.isi.edu/in-notes/fyi/fyi8.txt
- **CD-ROM:** \Appendx\App_H\others\fyi8.txt

**Description:** Contains information for system and network administrators to help them maintain the security of their systems.

*Internet FYI 34: Users’ Security Handbook*

- **URL:** ftp://ftp.isi.edu/in-notes/fyi/fyi34.txt
- **CD-ROM:** \Appendx\App_H\others\fyi34.txt

**Description:** Provides information on what users should do to help maintain computer security.
Other Standards

Directory Services Markup Language

**URL:** http://www.dsml.org/

**Description:** The directory services markup language (DSML) organization is defining XML for the interchange of directory data. This is expected to be a key directory interoperability standard.

Extensible Markup Language (XML)

**URL:** http://www.w3.org/XML/

**Description:** The starting page for the definition of XML (extensible markup language).

Java Naming and Directory Interface TM (JNDI)

**URL:** http://java.sun.com/products/jndi/index.html/

**CD-ROM:** A JNDI reference can be reached from the welcom.htm page.

**Description:** A directory-neutral interface for Java programs to access naming and directory services.

Vendor Information

alphaWorks

**URL:** http://www.alphaWorks.ibm.com/

**Description:** The IBM alphaWorks site. This site provides early access to various IBM technologies. The site currently has a large amount of XML tools available on it.

developerWorks

**URL:** http://www.ibm.com/developer/

**Description:** The IBM developerWorks site. This site currently has a large amount of XML tools available on it.
Directory Interoperability Forum Homepage

URL: http://www.directoryforum.org/

Description: The home page of the Directory Interoperability Forum. This industry group is focused on ensuring that directories from various vendors work together.

IBM SecureWay Directory


Description: The home page for IBM’s SecureWay Directory server.

IBM WebSphere

URL: http://www-4.ibm.com/software/webservers/

Description: The home page for IBM’s WebSphere application server.

Lotus Domino

URL: http://www.lotus.com/home.nsf/welcome/domino

Description: The home page for Lotus Domino groupware.

Microsoft Active Directory

URL: http://www.microsoft.com/windows/server/Overview/exploring/directory.asp

Description: The home page for Microsoft Active Directory.

MIT Kerberos

URL: http://web.mit.edu/kerberos/www/

Description: The home page for MIT’s Kerberos authentication service.

Novell NDS

URL: http://www.novell.com/products/nds/index.html

Description: The home page for Novell’s Directory Service.
OpenLDAP

URL: http://www.openldap.org/

Description: The home page of the OpenLDAP organization. This group is creating an open source implementation of a directory server.

Sun and Netscape iPlanet Directory


Description: The home page for the directory service from Netscape and Sun.
Appendix I

USEFUL ATTRIBUTE TYPES
AND OBJECT CLASSES

There are a number of object classes and attribute types to pay particular attention to in X.520 and in the IETF RFC 2252 defined set. The names of these object classes and attribute types are shown in Tables I.1 and I.2, respectively. LDIF-formatted files that define the object classes and attributes can be found on the accompanying CD.

Note that the IBM schema, described in Appendix G, contains all of these attribute types and object classes as well as many more. If an IBM SecureWay LDAP server is used, then the full IBM schema can be exploited.

Table I.1 Useful object class definitions

<table>
<thead>
<tr>
<th>Name</th>
<th>OID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>account</td>
<td>0.9.2342.19200300.100.4.5</td>
<td>Contains common account information.</td>
</tr>
<tr>
<td>applicationEntity</td>
<td>2.5.6.12</td>
<td>May be used to represent application entities in the directory. An applica-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>tion entity consists of those aspects of an application process pertinent to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OSI.</td>
</tr>
<tr>
<td>applicationProcess</td>
<td>2.5.6.11</td>
<td>May be used to define entries representing application processes in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>directory (see ISO 7498).</td>
</tr>
<tr>
<td>certificationAuthority</td>
<td>2.5.6.16</td>
<td>Used to store information about certificate authorities (CAs) in the directory.</td>
</tr>
<tr>
<td>certificationAuthority-V2</td>
<td>2.5.6.16.2</td>
<td>Represents information for a certification authority.</td>
</tr>
</tbody>
</table>

Table continued on next page.
Table I.1  Useful object class definitions (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>OID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>country</td>
<td>2.5.6.2</td>
<td>Defines entries that represent countries.</td>
</tr>
<tr>
<td>cRLDistributionPoint</td>
<td>2.5.6.19</td>
<td>Used to distribute revocation information</td>
</tr>
<tr>
<td>device</td>
<td>2.5.6.14</td>
<td>Used to store information about network devices, such as printers, in the directory</td>
</tr>
<tr>
<td>dmd</td>
<td>2.5.6.20</td>
<td>Defines a directory management domain.</td>
</tr>
<tr>
<td>dNSDomain</td>
<td>0.9.2342.19200300.100.4.15</td>
<td>Object class used as a subclass of domain to store DNS resource records in the directory</td>
</tr>
<tr>
<td>document</td>
<td>0.9.2342.19200300.100.4.6</td>
<td>Used to define entries that represent documents in the directory</td>
</tr>
<tr>
<td>documentSeries</td>
<td>0.9.2342.19200300.100.4.9</td>
<td>Used to define an entry that represents a series of documents</td>
</tr>
<tr>
<td>domain</td>
<td>0.9.2342.19200300.100.4.13</td>
<td>Object class used to define entries that represent DNS domains in the directory. The domainComponent attribute should be used for naming entries of this object class.</td>
</tr>
<tr>
<td>domainRelatedObject</td>
<td>0.9.2342.19200300.100.4.17</td>
<td>Object class used to define entries that represent a DNS domain equivalent to an X.500 domain, usually an organization or organizational unit</td>
</tr>
<tr>
<td>dSA</td>
<td>2.5.6.13</td>
<td>Used to define entries representing DSAs in the directory</td>
</tr>
<tr>
<td>friendlyCountry</td>
<td>0.9.2342.19200300.100.4.18</td>
<td>Object class used to define country entries in the directory tree. This object class is used to allow more user-friendly country names than those allowed by the country object class.</td>
</tr>
<tr>
<td>groupOfCertificates</td>
<td>2.16.840.1.113730.3.2.31</td>
<td>Represents a group of X.509 certificates</td>
</tr>
</tbody>
</table>
Table I.1  Useful object class definitions

<table>
<thead>
<tr>
<th>Name</th>
<th>OID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>groupOfNames</td>
<td>2.5.6.9</td>
<td>Defines entries for a group of names. Represents a list containing an unordered list of names.</td>
</tr>
<tr>
<td>groupOfUniqueNames</td>
<td>2.5.6.17</td>
<td>Defines entries for a group of unique names.</td>
</tr>
<tr>
<td>groupOfURLs</td>
<td>2.16.840.1.113730.3.2.33</td>
<td>Represents a group of URLs.</td>
</tr>
<tr>
<td>inetOrgPerson</td>
<td>2.16.840.1.113730.3.2.2</td>
<td>Defines entries representing people in an organization’s enterprise network.</td>
</tr>
<tr>
<td>javaContainer</td>
<td>1.3.18.0.2.6.161</td>
<td>Directory entry dedicated to storing a Java object</td>
</tr>
<tr>
<td>javaObject</td>
<td>1.3.18.0.2.6.162</td>
<td>Directory entries of this object class may contain a Java object instance or a Java object reference.</td>
</tr>
<tr>
<td>labeled URIObject</td>
<td>1.3.6.1.4.1.250.3.15</td>
<td>This object class can be added to existing directory objects to allow for inclusion of URI values. This approach does not preclude including the labeledURI attribute type directly in other object classes as appropriate.</td>
</tr>
<tr>
<td>liOrganization</td>
<td>1.3.6.1.4.1.1466.154.151.160.2</td>
<td>This structural class defines lightweight internet organizations; defined by Network Applications Consortium and The Open Group</td>
</tr>
<tr>
<td>liPerson</td>
<td>1.3.6.1.4.1.1466.154.151.160.1</td>
<td>This Network Application Consortium–defined structural class may be used to represent people and contains many commonly used organizational and residential attributes</td>
</tr>
<tr>
<td>locality</td>
<td>2.5.6.3</td>
<td>Defines entries that represent localities or geographic areas.</td>
</tr>
<tr>
<td>newPilotPerson</td>
<td>0.9.2342.19200300.100.4.4</td>
<td>Used as a subclass of person, to allow the use of a number of additional attributes to be assigned to entries of the person object class.</td>
</tr>
</tbody>
</table>

*Table continued on next page.*
<table>
<thead>
<tr>
<th>Name</th>
<th>OID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>organization</td>
<td>2.5.6.4</td>
<td>Defines entries that represent organizations. An organization is generally assumed to be a large, relatively static group within a larger corporation or enterprise.</td>
</tr>
<tr>
<td>organizationalPerson</td>
<td>2.5.6.7</td>
<td>Defines entries for people employed by or associated with an organization.</td>
</tr>
<tr>
<td>organizationalRole</td>
<td>2.5.6.8</td>
<td>Defines entries that represent roles held by people within an organization.</td>
</tr>
<tr>
<td>organizationalUnit</td>
<td>2.5.6.5</td>
<td>Defines entries that represent organizational units. An organizational unit is generally assumed to be a relatively static grouping within a larger organization.</td>
</tr>
<tr>
<td>person</td>
<td>2.5.6.6</td>
<td>Defines entries that generically represent people.</td>
</tr>
<tr>
<td>pilotObject</td>
<td>0.9.2342.19200300.100.4.3</td>
<td>Object class used as a subclass to allow additional attributes to be assigned to entries of all other object classes</td>
</tr>
<tr>
<td>residentialPerson</td>
<td>2.5.6.10</td>
<td>Represents people in a residential environment.</td>
</tr>
<tr>
<td>rFC822LocalPart</td>
<td>0.9.2342.19200300.100.4.14</td>
<td>Object class used to define entries that represent the local part of RFC822 mail addresses. The directory treats this part of an RFC822 address as a domain.</td>
</tr>
<tr>
<td>room</td>
<td>0.9.2342.19200300.100.4.7</td>
<td>Object class used to store information about a room in the directory.</td>
</tr>
<tr>
<td>strongAuthenticationUser</td>
<td>2.5.6.15</td>
<td>Used to store a user's certificate entry in the directory</td>
</tr>
<tr>
<td>top</td>
<td>2.5.6.0</td>
<td>Top of the object class hierarchy.</td>
</tr>
<tr>
<td>userSecurityInformation</td>
<td>2.5.6.18</td>
<td>User security information</td>
</tr>
</tbody>
</table>
### Table I.2  Useful attribute type definitions

<table>
<thead>
<tr>
<th>Name</th>
<th>OID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cn,commonName</td>
<td>2.5.4.3</td>
<td>This is the X.500 commonName attribute, which contains a name of an object. If the object corresponds to a person, it is typically the person's full name.</td>
</tr>
<tr>
<td>member</td>
<td>2.5.4.31</td>
<td>Identifies the distinguished names for each member of the group.</td>
</tr>
<tr>
<td>userPassword</td>
<td>2.5.4.35</td>
<td>Holds a password value for a distinguished name.</td>
</tr>
<tr>
<td>abstract</td>
<td>1.3.18.0.2.4.708</td>
<td>Provides an abstract of a document entry.</td>
</tr>
<tr>
<td>actionDate</td>
<td>1.3.18.0.2.4.73</td>
<td>The date of next occurrence, calculated from frequency each time the directory operation is run</td>
</tr>
<tr>
<td>associatedDomain</td>
<td>0.9.2342.19200300.100.1.37</td>
<td>Specifies a DNS domain associated with an object in the directory tree.</td>
</tr>
<tr>
<td>associatedName</td>
<td>0.9.2342.19200300.100.1.38</td>
<td>Specifies an entry in the organizational directory tree associated with a DNS domain.</td>
</tr>
<tr>
<td>audio</td>
<td>0.9.2342.19200300.100.1.55</td>
<td>Contains a sound file in binary format.</td>
</tr>
<tr>
<td>authorityRevocationList</td>
<td>2.5.4.38</td>
<td>Contains a list of CA certificates that have been revoked.</td>
</tr>
<tr>
<td>buildingName</td>
<td>0.9.2342.19200300.100.1.48</td>
<td>Defines the building name associated with the entry.</td>
</tr>
<tr>
<td>businessCategory</td>
<td>2.5.4.15</td>
<td>Describes the kind of business performed by an organization.</td>
</tr>
<tr>
<td>c,countryName</td>
<td>2.5.4.6</td>
<td>Contains a two-letter ISO 3166 country code (countryName).</td>
</tr>
<tr>
<td>caCertificate</td>
<td>2.5.4.37</td>
<td>Contains the CAs certificate.</td>
</tr>
<tr>
<td>carLicense</td>
<td>2.16.840.1.113730.3.1.1</td>
<td>Vehicle license plate tag</td>
</tr>
<tr>
<td>certificateRevocationList</td>
<td>2.5.4.39</td>
<td>Contains a list of revoked user certificates.</td>
</tr>
</tbody>
</table>

*Table continued on next page.*
### Table I.2 Useful attribute type definitions (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>OID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>changeLogMaximumAge</td>
<td>2.16.840.1.113730.3.1.200</td>
<td>Specifies, in seconds, the maximum age for entries in the change log. Entries are discarded once they become older than the maximum age specified.</td>
</tr>
<tr>
<td>changeLogMaximumSize</td>
<td>2.16.840.1.113730.3.1.201</td>
<td>Specifies, in bytes, the maximum size for the change log file. Entries are discarded from this file once it reaches the maximum size.</td>
</tr>
<tr>
<td>co,friendlyCountryName</td>
<td>0.9.2342.19200300.100.1.43</td>
<td>Contains the name of a country. Often, the country attribute is used to describe a two-character code for a country, and the friendlyCountryName attribute is used to describe the actual country name.</td>
</tr>
<tr>
<td>crossCertificatePair</td>
<td>2.5.4.40</td>
<td>The forward elements of the crossCertificatePair attribute of a CA's directory entry shall be used to store all except self-issued certificates issued to this CA. Optionally, the reverse elements of the crossCertificatePair attribute of a CA's directory entry may contain a subset of certificates issued by this CA to other CAs.</td>
</tr>
<tr>
<td>dc,domainComponent</td>
<td>0.9.2342.19200300.100.1.25</td>
<td>Specifies one component of a domain name.</td>
</tr>
<tr>
<td>deltaRevocationList</td>
<td>2.5.4.53</td>
<td>This attribute is to be stored and requested in the binary form, as deltaRevocationList;binary.</td>
</tr>
<tr>
<td>departmentNumber</td>
<td>2.16.840.1.113730.3.1.2</td>
<td>Identifies a department within an organization.</td>
</tr>
<tr>
<td>description</td>
<td>2.5.4.13</td>
<td>Attribute common to CIM and LDAP schema to provide lengthy description of a directory object entry</td>
</tr>
<tr>
<td>destinationIndicator</td>
<td>2.5.4.27</td>
<td>Used for the telegram service</td>
</tr>
<tr>
<td>Name</td>
<td>OID</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ditRedirect</td>
<td>0.9.2342.19200300.100.1.54</td>
<td>Used to indicate that the object described by one entry now has a newer entry in the directory tree. This attribute may be used when an individual's place of work changes and the individual acquires a new organizational DN.</td>
</tr>
<tr>
<td>dmdName</td>
<td>1.3.18.0.2.4.495</td>
<td>The value of this attribute specifies a directory management domain (DMD), the administrative authority that operates the directory server.</td>
</tr>
<tr>
<td>dn,distinguishedName</td>
<td>2.5.4.49</td>
<td>This attribute type is not used as the name of the object itself, but it is instead a base type from which attributes with DN syntax inherit. It is unlikely that values of this type itself will occur in an entry.</td>
</tr>
<tr>
<td>dnQualifier</td>
<td>2.5.4.46</td>
<td>Specifies disambiguating information to add to the relative distinguished name of an entry. Intended for use when merging data from multiple sources in order to prevent conflicts between entries that would otherwise have the same name. It is recommended that the value of the dnQualifier attribute be the same for all entries from a particular source.</td>
</tr>
<tr>
<td>dnsRecord</td>
<td>0.9.2342.19200300.100.1.26</td>
<td>Specifies DNS resource records, including type A (Address), type MX (Mail Exchange), type NS (Name Server), and type SOA (Start Of Authority) resource records.</td>
</tr>
<tr>
<td>documentAuthor</td>
<td>0.9.2342.19200300.100.1.14</td>
<td>Contains the distinguished name of the author of a document entry.</td>
</tr>
<tr>
<td>documentIdentifier</td>
<td>0.9.2342.19200300.100.1.11</td>
<td>Specifies a unique identifier for a document.</td>
</tr>
<tr>
<td>documentLocation</td>
<td>0.9.2342.19200300.100.1.15</td>
<td>Defines the location of the original copy of a document entry.</td>
</tr>
</tbody>
</table>

*Table continued on next page.*
<table>
<thead>
<tr>
<th>Name</th>
<th>OID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>documentPublisher</td>
<td>0.9.2342.19200300.100.1.56</td>
<td>The person and/or organization that published a document</td>
</tr>
<tr>
<td>documentTitle</td>
<td>0.9.2342.19200300.100.1.12</td>
<td>Contains the title of a document entry.</td>
</tr>
<tr>
<td>documentVersion</td>
<td>0.9.2342.19200300.100.1.13</td>
<td>Defines the version of a document entry.</td>
</tr>
<tr>
<td>drink, favouriteDrink</td>
<td>0.9.2342.19200300.100.1.5</td>
<td>Describes favorite drink of a person entry.</td>
</tr>
<tr>
<td>dsAQuality</td>
<td>0.9.2342.19200300.100.1.49</td>
<td>Server quality</td>
</tr>
<tr>
<td>emailFormat</td>
<td>1.3.18.0.2.4.78</td>
<td>The format used to build the e-mail address if possible.</td>
</tr>
<tr>
<td>employeeNumber</td>
<td>2.16.840.1.113730.3.1.3</td>
<td>Identifies the entry's employee number.</td>
</tr>
<tr>
<td>employeeType</td>
<td>2.16.840.1.113730.3.1.4</td>
<td>Identifies the entry's type of employment.</td>
</tr>
<tr>
<td>enhancedSearchGuide</td>
<td>2.5.4.47</td>
<td>This attribute is for use by X.500 clients in constructing search filters.</td>
</tr>
<tr>
<td>FacsimileTelephoneNumber, fax</td>
<td>2.5.4.23</td>
<td>Identifies the fax number at which the entry can be reached.</td>
</tr>
<tr>
<td>geographicalCoverage</td>
<td>1.3.18.0.2.4.59</td>
<td>A list of geographical locations in which an organization’s products or services are offered</td>
</tr>
<tr>
<td>givenName</td>
<td>2.5.4.42</td>
<td>Used to hold the part of a person’s name that is not their surname nor middle name.</td>
</tr>
<tr>
<td>homeFax</td>
<td>1.3.6.1.4.1.1466.101.120.31</td>
<td>Identifies the entry's home fax number.</td>
</tr>
<tr>
<td>homePhone</td>
<td>0.9.2342.19200300.100.1.20</td>
<td>Identifies the entry's home phone number.</td>
</tr>
<tr>
<td>homePostalAddress</td>
<td>0.9.2342.19200300.100.1.39</td>
<td>Identifies the entry's home mailing address. This field is intended to include multiple lines, but each line within the entry should be separated by a dollar sign ($). To represent an actual dollar sign ($) or backslash () within this text, use the escaped hex values \24 and \5c, respectively.</td>
</tr>
<tr>
<td>Name</td>
<td>OID</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>host, hostName</td>
<td>1.3.18.0.2.4.486</td>
<td>Generally, the DNS hostname of a computer system. When used in non-TCP/IP environments for naming a computer system, the nameFormat attribute is used to identify the type of name used.</td>
</tr>
<tr>
<td>houseIdentifier</td>
<td>2.5.4.51</td>
<td>Used to identify a building within a location</td>
</tr>
<tr>
<td>info</td>
<td>0.9.2342.19200300.100.1.4</td>
<td>Specifies any general information pertinent to an object. It is recommended that this attribute type be avoided, and that specific requirements are met by other (possibly additional) attribute types.</td>
</tr>
<tr>
<td>initials</td>
<td>2.5.4.43</td>
<td>Contains the initials of some or all of an individual's names, but not the surname(s).</td>
</tr>
<tr>
<td>internationallySDNNumber</td>
<td>2.5.4.25</td>
<td>Contains the ISDN number of the entry. This is in the internationally agreed format for ISDN addresses given in CCITT Rec. E. 164.</td>
</tr>
<tr>
<td>janetMailbox</td>
<td>0.9.2342.19200300.100.1.46</td>
<td>Specifies an e-mail address. This attribute is intended for the convenience of UK users unfamiliar with rfc822 mail addresses. Entries using this attribute must also include an rfc822Mailbox attribute.</td>
</tr>
<tr>
<td>javaClassName</td>
<td>1.3.18.0.2.4.146</td>
<td>Indicates the name of the Java class.</td>
</tr>
<tr>
<td>javaSerializedObject</td>
<td>1.3.18.0.2.4.707</td>
<td>Represents a Java serialized object.</td>
</tr>
<tr>
<td>jpegPhoto</td>
<td>0.9.2342.19200300.100.1.60</td>
<td>Contains a JPEG photo of the entry.</td>
</tr>
<tr>
<td>keywords</td>
<td>1.3.18.0.2.4.500</td>
<td>Contains keywords for the entry.</td>
</tr>
<tr>
<td>knowledgeInformation</td>
<td>2.5.4.2</td>
<td>This attribute is no longer used.</td>
</tr>
<tr>
<td>localityName</td>
<td>2.5.4.7</td>
<td>Contains the name of a locality, such as a city, county, or other geographic region (localityName).</td>
</tr>
</tbody>
</table>
### Table I.2 Useful attribute type definitions (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>OID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>labeledURI</td>
<td>1.3.6.1.4.1.250.1.57</td>
<td>Uniform resource identifier with optional label as defined in RFC2079.</td>
</tr>
<tr>
<td>mail,rfc822mailbox</td>
<td>0.9.2342.19200300.100.1.3</td>
<td>Identifies a user's primary e-mail address (the e-mail address retrieved and displayed by white pages lookup applications).</td>
</tr>
<tr>
<td>mailPreferenceOption</td>
<td>0.9.2342.19200300.100.1.47</td>
<td>Indicates a preference for inclusion of user names on mailing lists (electronic or physical).</td>
</tr>
<tr>
<td>manager</td>
<td>0.9.2342.19200300.100.1.10</td>
<td>Identifies the distinguished name of the entry's manager.</td>
</tr>
<tr>
<td>MemberCertificateDescription</td>
<td>2.16.840.1.113730.3.1.1.199</td>
<td>Description of groupOfCertificates</td>
</tr>
<tr>
<td>membership</td>
<td>1.3.18.0.2.4.62</td>
<td>Specifies an organization's membership in a professional organization.</td>
</tr>
<tr>
<td>memberURL</td>
<td>2.16.840.1.113730.3.1.198</td>
<td>Identifies a URL associated with each member of a group. Any type of labeled URL can be used.</td>
</tr>
<tr>
<td>middleName</td>
<td>1.3.6.1.4.1.1466.101.120.34</td>
<td>Identifies the entry's middle name.</td>
</tr>
<tr>
<td>mobile,mobileTelephoneNumber</td>
<td>0.9.2342.19200300.100.1.41</td>
<td>Identifies the entry's mobile or cellular phone number.</td>
</tr>
<tr>
<td>name</td>
<td>2.5.4.41</td>
<td>Attribute supertype from which string attribute types typically used for naming may be formed. It is unlikely that values of this type itself will occur in an entry.</td>
</tr>
<tr>
<td>o,organizationName,organization</td>
<td>2.5.4.10</td>
<td>Contains the name of an organization (organizationName).</td>
</tr>
<tr>
<td>organizationalStatus</td>
<td>0.9.2342.19200300.100.1.45</td>
<td>Specifies a category by which a person is often referred to in an organization.</td>
</tr>
<tr>
<td>otherMailbox</td>
<td>0.9.2342.19200300.100.1.22</td>
<td>Specifies values for electronic mailbox types other than X.400 and rfc822.</td>
</tr>
<tr>
<td>ou,organizationalUnit</td>
<td>2.5.4.11</td>
<td>Contains the name of an organizational unit.</td>
</tr>
<tr>
<td>Name</td>
<td>OID</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>owner</td>
<td>2.5.4.32</td>
<td>Identifies the distinguished name of the person responsible for the entry.</td>
</tr>
<tr>
<td>pager,pagerTelephoneNumber</td>
<td>0.9.2342.19200300.100.1.42</td>
<td>Identifies the entry's pager phone number.</td>
</tr>
<tr>
<td>passwordMaxAge</td>
<td>2.16.840.1.113730.3.1.97</td>
<td>Specifies, in seconds, the period of time passwords can be used before they expire.</td>
</tr>
<tr>
<td>passwordMinAge</td>
<td>1.3.18.0.2.4.465</td>
<td>Specifies, in seconds, the period of time a password must be in effect before a user can change it.</td>
</tr>
<tr>
<td>passwordMinLength</td>
<td>2.16.840.1.113730.3.1.99</td>
<td>Specifies the minimum number of characters required for a user's password.</td>
</tr>
<tr>
<td>personalSignature</td>
<td>0.9.2342.19200300.100.1.53</td>
<td>A signature file, in binary format, for the entry.</td>
</tr>
<tr>
<td>personalTitle</td>
<td>0.9.2342.19200300.100.1.40</td>
<td>Specifies a personal title for a person. Examples of personal titles are Mr, Ms, Dr, Prof, and Rev.</td>
</tr>
<tr>
<td>photo</td>
<td>0.9.2342.19200300.100.1.7</td>
<td>Contains a photo, in binary form, of the entry.</td>
</tr>
<tr>
<td>PhysicalDeliveryOfficeName</td>
<td>2.5.4.19</td>
<td>Physical delivery office number</td>
</tr>
<tr>
<td>postalAddress</td>
<td>2.5.4.16</td>
<td>Identifies the entry's mailing address. This field is intended to include multiple lines. When represented in LDIF format, each line should be separated by a dollar sign ($).</td>
</tr>
<tr>
<td>postalCode</td>
<td>2.5.4.17</td>
<td>Postal code</td>
</tr>
<tr>
<td>postOfficeBox</td>
<td>2.5.4.18</td>
<td>Post office box number</td>
</tr>
<tr>
<td>preferredDeliveryMethod</td>
<td>2.5.4.28</td>
<td>Identifies the entry's preferred contact or delivery method.</td>
</tr>
<tr>
<td>preferredLanguage</td>
<td>2.16.840.1.113730.3.1.39</td>
<td>A person's preferred written or spoken language</td>
</tr>
<tr>
<td>presentationAddress</td>
<td>2.5.4.29</td>
<td>Contains an OSI presentation address.</td>
</tr>
</tbody>
</table>
### Table I.2  Useful attribute type definitions (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>OID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocolInformation</td>
<td>2.5.4.48</td>
<td>Used in conjunction with the presentationAddress attribute, to provide additional information to the OSI network service.</td>
</tr>
<tr>
<td>registeredAddress</td>
<td>2.5.4.26</td>
<td>Contains a postal address for receiving telegrams or expedited documents. The recipient's signature is usually required on delivery.</td>
</tr>
<tr>
<td>roleOccupant</td>
<td>2.5.4.33</td>
<td>Contains the distinguished name of the person acting in the role defined in the organizationalRole entry.</td>
</tr>
<tr>
<td>roomNumber</td>
<td>0.9.2342.19200300.100.1.6</td>
<td>Specifies the room number of an object. Note that the commonName attribute should be used for naming room objects.</td>
</tr>
<tr>
<td>searchGuide</td>
<td>2.5.4.14</td>
<td>For use by X.500 clients in constructing search filters. It is obsoleted by enhancedSearchGuide.</td>
</tr>
<tr>
<td>secretary</td>
<td>0.9.2342.19200300.100.1.21</td>
<td>Identifies the entry's secretary or administrative assistant.</td>
</tr>
<tr>
<td>seeAlso</td>
<td>2.5.4.34</td>
<td>Identifies another directory server entry that may contain information related to this entry.</td>
</tr>
<tr>
<td>serialNumber</td>
<td>2.5.4.5</td>
<td>Contains the serial number of a device.</td>
</tr>
<tr>
<td>sn,surName</td>
<td>2.5.4.4</td>
<td>Contains the family name of a person.</td>
</tr>
<tr>
<td>st,stateOrProvince</td>
<td>2.5.4.8</td>
<td>Contains the full name of a state or province (stateOrProvinceName).</td>
</tr>
<tr>
<td>street,streetAddress</td>
<td>2.5.4.9</td>
<td>Contains the physical address of the object to which the entry corresponds, such as an address for package delivery (streetAddress).</td>
</tr>
</tbody>
</table>

Table continued on next page.
### Table I.2  Useful attribute type definitions

<table>
<thead>
<tr>
<th>Name</th>
<th>OID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>supportedAlgorithms</td>
<td>2.5.4.52</td>
<td>To be stored and requested in the binary form, as supportedAlgorithm;binary</td>
</tr>
<tr>
<td>supportedApplicationContext</td>
<td>2.5.4.30</td>
<td>Contains the identifiers of OSI application contexts.</td>
</tr>
<tr>
<td>telephoneNumber</td>
<td>2.5.4.20</td>
<td>Telephone number</td>
</tr>
<tr>
<td>teletexTerminalIdentifier</td>
<td>2.5.4.22</td>
<td>Teletex terminal identifier</td>
</tr>
<tr>
<td>TelexNumber, telexeNumber</td>
<td>2.5.4.21</td>
<td>Telex number</td>
</tr>
<tr>
<td>textEncodedOrAddress</td>
<td>0.9.2342.19200300.100.1.2</td>
<td>Defines the text-encoded Originator/Recipient (X.400) address of the entry as defined in RFC987.</td>
</tr>
<tr>
<td>thumbNailLogo</td>
<td>2.16.128.113533.1.1400.2</td>
<td>Thumbnail logo</td>
</tr>
<tr>
<td>thumbNailPhoto</td>
<td>2.16.128.113533.1.1400.1</td>
<td>Thumbnail photo</td>
</tr>
<tr>
<td>title</td>
<td>2.5.4.12</td>
<td>Contains the title, such as vice president, of a person in their organizational context. The personalTitle attribute would be used for a person's title independent of their job function.</td>
</tr>
<tr>
<td>ttl,timeToLive</td>
<td>1.3.6.1.4.1.250.1.60</td>
<td>Time-to-live attribute</td>
</tr>
<tr>
<td>TypelessRDN</td>
<td>1.3.22.1.2.2</td>
<td>Typeless RDN as defined by OSF</td>
</tr>
<tr>
<td>uid</td>
<td>0.9.2342.19200300.100.1.1</td>
<td>Typically a user shortname or user ID.</td>
</tr>
<tr>
<td>uniqueIdentifier</td>
<td>0.9.2342.19200300.100.1.44</td>
<td>Identifies a specific item used to distinguish between two entries when a distinguished name has been re-used. This attribute is intended to detect instances of a reference to a distinguished name that has been deleted. This attribute is assigned by the server.</td>
</tr>
</tbody>
</table>
Be sure to use these object classes and attribute types as the starting point for defining new schemas. Consider using the more complete and CIM-based IBM schema as well. The object classes and attribute types in X.520, IETF RFC 2252, and the IBM schema serve as a strong starting point for defining new schemas.

Table 1.2 Useful attribute type definitions (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>OID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniqueMember</td>
<td>2.5.4.50</td>
<td>Identifies a group of names associated with an entry where each name was given a uniqueIdentifier to ensure its uniqueness. A value for the uniqueMember attribute is a DN followed by the uniqueIdentifier.</td>
</tr>
<tr>
<td>userCertificate</td>
<td>2.5.4.36</td>
<td>Used to represent certificates from one or more certification authorities representing a user</td>
</tr>
<tr>
<td>userClass</td>
<td>0.9.2342.19200300.100.1.8</td>
<td>Specifies a category of computer user. The semantics of this attribute are arbitrary.</td>
</tr>
<tr>
<td>userSMIMECertificate</td>
<td>2.16.840.1.113730.3.1.40</td>
<td>Signed message used to support S/MIME</td>
</tr>
<tr>
<td>x121address</td>
<td>2.5.4.24</td>
<td>Defines the X.121 address of the entry.</td>
</tr>
<tr>
<td>x500UniqueIdentifier</td>
<td>2.5.4.45</td>
<td>Used to distinguish between objects when a distinguished name has been re-used. This is a different attribute type from both the uid and uniqueIdentifier types.</td>
</tr>
</tbody>
</table>
Using the CD

The CD accompanying this book contains sample code, supplementary write-ups, and in some cases information from other sources. It should be very easy to browse. The CD is laid out with hyperlinks from a main page to a readme.htm page for each chapter or appendix that has CD material (and these often link off to readme.htm pages for samples or other material).

CD Contents

Figure J.1 shows what to expect when using the CD. Also included on the CD (the “Other Supplementary” material referenced in Figure J.1) are the following.

*Understanding LDAP*, SG24-4986-00, an IBM Redbook including an LDAP API description and some example programs.

*LDAP Implementation Cookbook*, SG24-5110-00, an IBM Redbook focusing on how to implement LDAP directory services in your organization. This book contains good information on planning for and rolling out directory services.

*Ready for e-business: OS/390 Security Server Enhancements*, SG24-5158-00, an IBM Redbook discussing how to do business on the Internet. It includes sections on the OS/390 directory server and background on directory security services.

*The Java Naming and Directory Interfaces (JNDI) Specification*, a complete specification for directory services in Java, included by permission of Sun Microsystems.

Several OS/390 directory specific publications. These give you a taste of running a real directory service in production and what it takes to plan for and program it.

A link to information on installing the IBM SecureWay LDAP Directory for Microsoft Windows NT and supporting products (contained on the CD—you are not required to download anything). To install the directory server, you will
be required to agree to the licensing terms, which include restricting use to
development only (the server may not be used for any production purpose). In
addition to the SecureWay server; sample programs, documentation, and a cli-
ent SDK (for C and Java) are also included (again with licensing terms). See the
Installation and Configuration Guide for details on getting the directory running.
This is a directory product (not just a tool included with the book). Sorry, there
is no “one-click” installation procedure.

You will also find this book on the CD. This may make it convenient to take the CD
with you on the road, instead of or in addition to the printed copy.

**Using the CD**

On Windows systems, your default browser should come up when you insert the
CD, provided autorun is enabled (it usually is). If for whatever reason welcom.htm
does not come up when you insert the CD (for example, you are not running Win-
dows or autorun is disabled), start your favorite browser and point it at the wel-

---

**Figure J.1** Simplified CD contents
come.htm file. In Windows you can also just double-click on the welcom.htm file from Explorer to start your default browser.

**Startup.exe**

The curious can see that a program called startup.exe is run by the autorun.inf script in the CD root. The source for this program is in a subdirectory of the auto directory. If you look, you will see that startup.cpp contains only one line of useful code. That one line will cause your default browser to start and load the page listed on the autorun.inf startup command line (welcom.htm). If we were bad people (we are not), the startup.exe program could do bad things to your computer. It’s good to remember that and decline to insert CDs of unknown origin, unless you first disable the autorun feature (technically, autorun can work with diskettes or any other removable media as well, but no one will have that enabled by default).

**Browser Versions, Executable Programs, and File Formats**

Depending on your browser, you may find that links to executables don’t start the executables. If you want to start a program and your browser does not give you an option to do so, you can start the program manually (in Windows, for example, double-click on the .exe program from the Windows Explorer or start it from the Run menu).

The same may be true for file formats, such as PDF (Adobe Acrobat). Some material on the CD is in PDF files. Your browser may browse these files just fine, or you may have the program to view the file. In the case of PDF, you can download the free Adobe Acrobat viewer from http://www.adobe.com (a version of the Adobe Acrobat Viewer is on the CD).

**Executable Programs and Your Operating System**

The CD with this book is largely geared toward Windows users. Executable programs on the CD are Microsoft Windows executables. For example, the install program for the IBM SecureWay LDAP Directory is a Windows program.

If you would like a trial version of the IBM SecureWay LDAP Server for another platform, please visit http://www.software.ibm.com and search for SecureWay (or LDAP).
Sample Programs

As mentioned in the preface, example programs are provided on the CD. Each example was built and run on one operating system. Some examples were built with Microsoft Windows 2000, others with IBM OS/390, and still others with Linux.

No matter where an example was built and tested, the directory portion of the code will be of interest to readers of this book. In many cases, there are differences among operating systems that prevent a program written for one OS from running on another OS. In some cases, the differences are trivial and in other cases the differences are not trivial. Add to this that some directory services support different features, and that directory services can be configured a variety of ways. The result is that programs require porting.

There is usually a separation between the code that uses the directory services (using LDAP C or JNDI) and other code, such as for GUI rendering. Oftentimes, the code using directory services is of the most interest to readers of this book. So although the examples may be written on different operating systems, you should not have too much trouble seeing the relevant portions and porting those to whatever operating system you wish to use.

Requirements for Running the Samples

To make it easier for you to get the at-a-glance description of each sample and what is involved in porting it to other programs, a Quick Summary table is provided for each sample program on the CD. When you browse the CD, you will see links to the tables. Table J.1 is an example Quick Summary table from Chapter 5.

With only a few exceptions, the sample programs require, at a minimum, that you have access to a directory service. In addition, you will have to make modifications to the samples to run them on other than the tested platforms listed in the Quick Summary. Sometimes, the modifications will be slight and other times the modifications will be major, depending on your chosen directory service, language, and configuration options. The Quick Summary for each sample is a good guide if you want to port the sample to another platform or directory service.

Program Environment

An alternative would be to provide all samples in one language and on one operating system. This would make the book an advertisement for the one chosen operating system, rather than an exploration of enterprise computing with directories.
Why not write all the samples in one language, such as Java? While Java is extremely popular and growing moreso, many programmers and businesses still use other languages. This book provides a broad overview of those languages, while focusing mostly on enterprise software with LDAP C or JNDI. The main idea is not the language that is used, but enterprise computing and directories. Examples of several APIs are included in various samples (C, Java, Perl, and Visual Basic, with most being written in C or Java).

<table>
<thead>
<tr>
<th>Table J.1  Quick Sample Summary</th>
</tr>
</thead>
</table>
| **Chapter:** 5  
**Sample Name:** Locatew |
| **Purpose of Sample** | Show how to locate directory servers, keep a private cache of server locations for use in subsequent program invocations. |
| **Programming Language(s)** | C |
| **Wire protocol(s)** | LDAP |
| **Platform(s) tested on** | Windows 2000 |
| **Ease of porting to other platforms** | 1 for LDAP C code, 5 for GUI code (rewrite for any platform other than Windows), 4 for platform specific cache (sample uses Windows registry). |
| **Hints for porting to other platforms** | The LDAP C code is in file locatew.cpp, the GUI code is contained in separate files. |
| **Directory Service(s) tested with** | Microsoft Active Directory |
| **Hints for using other directory services** | Should be easy to do, except you may have to change directory server names, supply user credentials (for example, userid/password), and possibly map the serviceConnectionPoint objectclass to a class in your chosen directory. |
Commonly used terms and acronyms are explained here. Some of these have URL references in Appendix H for more information.

**Access control**— The process of controlling user access to resources. Usually implemented using access control lists.

**ACID**— Atomicity, consistency, isolation, and durability describe robust data characteristics, such as implemented in recoverable databases.

**ACL**— Access control lists. A list that identifies users or groups that are to be permitted or denied access to a resource.

**ActiveX**— Microsoft’s term for a type of component based on COM.

**ADSI**— Active directory services interface. A directory application programming framework defined by Microsoft.

**API**— Application programming interface. A structured set of rules for accessing a programming library or service.

**Application directory**— Information in a directory service that is specific to a particular application or application configuration.

**Application server**— A service that facilitates three-tier applications. Usually associated with a Web server that handles interactions with a client browser. Application servers often facilitate the maintenance of session state in the inherently stateless Web environment.

**ASCII**— American Standard Character Information Interchange. An 8-bit scheme for encoding characters.

**ASN.1**— Abstract Syntax Notation 1. A declarative language for describing data structures. ASN.1 is defined by the X.208 standard.
**ASP**—Active server pages. A server-side middle-tier programming model featuring scripts embedded in HTML text. Developed by Microsoft but also appearing in various Unix guises.

**Attribute type**—A piece of information, stored within an entry which is stored in a directory service. Attribute types within entries can contain multiple values. Every directory entry (or object) is composed of attribute-value pairs.

**Attribute**—See Attribute type.

**AVA**—Attribute value assertion. A particular attribute type equals a value combination. AVAs are usually used in creating relative distinguished names which, in turn, are used to create distinguished names for entries in a directory.

**Authentication**—The action of determining whether a presented identity is valid and being presented by its owner. (For example, showing your driver's license to a police officer.)

**BER**—Basic encoding rules. A set of rules for formulating wire-format byte streams for data structures that are defined using ASN.1 notation. BER formats are defined by the X.209 standard.

**B2B**—Business-to-business. A popular acronym for describing cooperative communications over the Internet between businesses in order to streamline current processes.

**CCITT**—International Telegraph and Telephone Consultative Committee. Now known as the ITU-T.

**CDS**—Cell directory services. The directory service that is integrated into DCE.

**CGI**—Common gateway interface. A common Web programming interface. CGI does not scale well because CGI creates a process for each user interaction. Scalable outgrowths of CGI feature pools of reusable processes or threads.

**CICS**—Customer Information and Control System. An IBM transaction processing and database system.

**CIM**—Component information model. A standard architecture and data model for interoperable systems management.

**Client**—An executable part of a distributed application that makes requests of a service to carry out part of the application’s function. A client may possibly interact with a single user.

**Client-server**—A distributed application programming paradigm distinguished by partitioning the application into a service instance shared by many remote client instances. There is usually a user associated with each client instance.
Cluster—A group of computers that appear as one. Fail-over clusters enable the workload of a failing computer to be taken on by the other members of the cluster. Load-balancing clusters allocate workload evenly among members. Both types of behaviors may coexist in one cluster.

COM—Microsoft Component Object Model. A COM component is a language-neutral entity having well-structured programming interfaces available for re-use within a system or network.


CORBA—Common object request broker architecture. A distributed-object architecture defined by the OMG.

Corporate directory—Information in a directory service that is applicable across an organization or needs to be made accessible across an entire organization. An example of this type of information is a corporate phone book.

COSNaming—(also OMG Name Service). The naming service defined by the OMG as part of the CORBA model for locating distributed objects.

Daemon—Another name for a background service. For example, DNS is provided by the named daemon. Pronounced “demon.”

DAP—Directory access protocol. The protocol for communication between DUAs and DSAs in the X.500 directory model.

DBMS—Database management system. Often has an SQL interface and participates in distributed transactions.

DCE—Distributed computing environment.

DCOM—Distributed component object model. The wire protocol for Microsoft’s COM and COM+ components. Based on an extension of DCE/RPC.

DDNS—Dynamic DNS. A DNS server that is updateable without restarting the server.

DEN—Directory enabled networking. An industry initiative to promote interoperable network configuration and management using a directory service.

Directory entry—An entry in the directory name space. An entry contains a set of attribute types, each of which may be assigned one or more values.

Directory hierarchy—See Directory information tree.
Directory information tree—Describes the hierarchical structure of a directory name space.

Directory server—A daemon, typically running on a server, which responds to directory clients’ requests for information stored in the directory.

Directory service—The collection of directory clients and servers that together provide access to information in the directory. Individual directory servers hold portions of the name space. The directory service provides access to the full name space.

Distinguished name—Each entry in a directory name space has a unique name, called its distinguished name.

Distributed transaction controller—A service that orchestrates a two-phase commit protocol among network-distributed parties that are members of an ACID transaction. EJB, COM+, CICS, and Tuxedo use the services of a DTC.

DIT—See Directory information tree.

DIT structure rules—rules defined in a directory server that control what types (object classes) of entries can be defined in what portions of the name space.

DMTF—Distributed Management Task Force. A standards body concerned with defining standards for interoperable systems management.

DN—See Distinguished name.

dn-pointer—An attribute value within an entry in a directory that is the distinguished name of another entry in the directory.

DNS—Domain name service. Defined by IETF RFCs.

Domino—IBM Lotus Domino is a groupware system. Lotus Notes is the client; Lotus Domino is the server.

DSA—Directory service agent. The server function in the X.500 directory model.

DSML—Directory services markup language. An XML DTD for describing information stored in a directory.

DTD—Document type definition. A definition of a set of XML tags and their relationships.

DUA—Directory user agent. The client side of the X.500 directory model.

EBCDIC—Extended binary coded decimal interchange code. A character code format that is used in IBM OS/390 and OS/400 operating systems.
**ECC**— Error correcting code

**EJB**— Enterprise Java beans. A component manager manages instances of components. Instances may participate in distributed transactions in a scalable manner.

**EJS**— Enterprise Java server. A functional unit of EJB that manages distributed transactions among component instances.

**Enterprise**— As used in this book, an enterprise might be a company, government agency, or other organization.

**Enterprise directory**— A directory service that serves the needs of a corporate directory, system directories, and application directories.

**Enterprise software**— As used in this book, software critical to the operation of an enterprise.

**Entry**— Each “thing” in a directory service is an entry. Each entry is named (see Distinguished name) and contains attribute-value pairs.

**Framework**— As applied to programming interfaces, a set of objects and interfaces that, together, form a programming environment to use in creating programs.

**Generic directory service**— see Enterprise directory.

**GSS-API**— Generic security services application programming interface. An API for accessing security services for authentication, data confidentiality, and data integrity in a service independent fashion. Currently defined by the IETF.

**GUID**— Globally unique identifier. 128-bit long computer-generated unique values. The algorithm used to generate these ensures that each possible value will not be generated more than once.

**HTML**— Hypertext markup language. A document tag language derived from SGML. HTML, unlike SGML, is not dynamically extensible.

**HTTP**— Hypertext transfer protocol. The document protocol used between Web browsers and Web services.

**IA5**— a character set composed of the first 128 characters of the ASCII character set. IA5 data is sometimes referred to as 7-bit ASCII or invariant ASCII.

**IANA**— Internet address name authority. The original control point for managing DNS names and addresses for the Internet.

**IDL**— Interface definition language

**IEEE**— Institute of Electrical and Electronics Engineers
IETF—Internet Engineering Task Force. A group that defines Internet standards.

IIOP—Internet Inter-ORB Protocol. The wire protocol used by Common Object Request Broker component instances.

IPX/SP—A networking protocol developed by Novell for establishing Netware networks.

ITU-T—International Telecommunication Union—Telecommunication Standardization Bureau.

Kerberos—Project Athena at MIT established the Kerberos secret key security algorithms which are today specified by IETF RFCs.

J2EE—Java 2 Enterprise Edition

JLDAP—Java directory access protocol. A Java programming interface being worked through the IETF.

JNDI—Java naming and directory interfaces. A Sun Java standard.

JTA/JTS—Java transaction API, Java transaction service

LDAP, LDAP Version 2, LDAP Version 3—Lightweight directory access protocol (versions 2 and 3). IETF RFCs define these.

LDAP server—A directory server that uses the LDAP protocol as the mechanism for accepting and emitting information stored in the directory.

LDIF—LDAP data interchange format. A portable, flat-file format for transferring LDAP directory content from one location to another.

LU—Logical unit. An entity that defines end-points for conversations over SNA networks.

Matching rule—Each attribute type has a set of matching rules that are applied to its values. The set of matching rules allowed or an attribute type is dependent on the attribute type’s associated syntax.

Metadirectory—A service that makes several disparate directories or schemas appear as one. An enabler for synchronization among directory services.

MOM—Message-oriented middleware. Examples are MQSeries and MSMQ.

MQ Series—An IBM message queuing system.

MSMQ—Microsoft Message Queuing
**MTA**—Mail Transfer Agent

**Name space**—The hierarchy of distinguished names that is served by the directory service. A directory server manages information in a portion of the overall name space, while the directory service provides access to the full name space.

**Naming constraints**—Rules defined in a directory server that control what attribute types can and/or must be used in creating distinguished names for entries in a directory.

**NCP**—Novell core protocol. The protocol used between Novell clients and Novell Netware servers.

**NDAP**—Novell Directory Access Protocol. The protocol used between Novell NDS clients and Novell NDS servers for accessing and modifying directory information.

**n-tier**—An application structuring model in which the application is distributed across multiple execution tiers. Most commonly used in describing applications with at least three tiers of applications—client, middle-tier, or Web application server—and third-tier database servers.

**Object**—In the context of directories, see entry.

**Objectclass (and object class)**—A schema element that defines the format of a particular kind of directory entry. For example, if the user is a valid directory entry then every person in the corporation has a user entry in the directory. User itself is defined by an objectclass that describes what attributes it must or might contain, as well as where in the DIT user entries can be located.

**Object class hierarchy**—Object classes can be derived from one another. This forms an inheritance hierarchy.

**Object identifier (OID)**—Dotted decimal string of numbers that uniquely identifies elements of a directory service schema: syntaxes, matching rules, attribute types, and object classes. Also used in defining LDAP Version 3 extended operations and controls.


**OSI**—Open systems interconnection. A layered networking protocol that was an alternative to SNA and TCP/IP.

**Partitioning**—Directory partitions are subsets of a larger directory name space.

**Property**—In the context of directory services, see Attribute.
Protocol— A set of rules for carrying out correspondence between entities

RADIUS (and RADIUS server)— Remote authentication dial-in user service

RAID— Redundant array of inexpensive devices. May be used to eliminate a potential single point of failure among devices.

RASSS— Reliability, availability, serviceability, scalability, and security.

RDN— Relative distinguished name. The portion of a distinguished name for an entry that is distinct from the distinguished name of the entry’s parent.

Replication— Directory replicas are copies. Generally applies to copies residing in identical directory services. Also refers to DBMS replicas.

Resource monitor— A functional unit provided by a resource, such as a DBMS, to enable participation in a distributed ACID transaction orchestrated by a DTC.

RFC— Request for comments. These are IETF standards.


RMI/IIO P— RMI protocol used within CORBA IIOP protocol.

RPC— Remote procedure call. Defined in DCE. Adapted by Microsoft’s DCOM wire protocol.

SASL— Simple authentication and security layer. An IETF standard for adding security to an existing application protocol. This is one of the security measures supported by the LDAP protocol.

Schema— The definition of what can go in a directory service.

Schema package— A subset of an entire schema that is defined in a directory server. A schema package allows for the identification of a subset of object class and attribute type definitions.

Script— A late-bound programming language, usually not needing an explicit compilation step. A few examples: VBScript, JavaScript, Perl, Tcl, Python, Rexx, and the Unix shells.

Service— A software server or daemon available for use by multiple networked clients.

Server— A hardware machine, often used to host services or daemons.

SGML— Standard generalized markup language. A markup language distinguished by a metagrammar allowing arbitrary document type definitions. The basis for XML.
**SLP**—Service locator protocol. An IETF standard for finding advertised services.

**SMTP**—Simple mail transport protocol

**SMP**—Symmetric multiprocessors. A computer with more than one processor. Many modern operating systems support SMP by allocating threads of execution among processors or dedicating each to a processor. Windows 9x isn’t an example of an SMP-capable system. Windows 2000 and Linux are.

**SNA**—Systems network architecture. A networking protocol invented by IBM to carry many different types of data communications.

**SOAP**—Simple object access protocol. A cross-vendor data marshalling specification based on XML. A standard has not been ratified. Alpha code working with Perl is available. Discussed as a possible replacement for DCOM.

**SQL**—Structured query language. A standard declarative data definition and data manipulation language.

**SRV record**—Service record. An information type that can be stored in DNS. Used to store extended connection information for servers running in a DNS domain that handle well-known services.

**SSL**—Secure sockets layer. A general protocol for adding security to socket-based applications. This is one of the security protocols supported by the LDAP protocol standards.

**SSPI**—System security provider interfaces. Windows programming interface for authentication, data confidentiality, and data integrity. Roughly equivalent to the GSS-API.

**Synchronization**—The process of keeping equivalent entries consistent across mixed directory services or a mix of schemas.

**Syntax**—Each attribute allowed in a directory service conforms to a particular syntax. For example, an emailID attribute might be defined as containing values that are Unicode strings. That is, the emailID is of Unicode string syntax.

**System directory**—Information in a directory service that is specific to a particular hardware system or operating system instance.

**TCP/IP**—Transmission control protocol/Internet protocol. The de facto standard for data communications worldwide.

**Thick client**—A client-server application having most of the program logic in the client. The server typically is a DBMS.
**Thin client**— A client-server application where most of the logic is in the server. Also refers to minimal computing machines having little or no disk storage.

**Three-tier**— A distributed application divided into three parts: a user GUI, a remote client, and a backend data store. The client sits in the middle tier. It interacts with the user GUI and a remote data store. The GUI is usually browser-based. The middle tier resides in a Web server or an application server bound to a Web server.

**Transaction**— A logical unit of work. Informally used to refer to any work done in response to a single user request. Transaction systems often are used to implement the ACID properties.

**Two-tier**— Another name for a client-server application

**TXT record**— Text record. An information type that can be stored in DNS. Used to store extra information in DNS that is associated with the DNS name.

**Unicode**— A mechanism to encode most of the world’s languages using 16 bits. The Unicode standard is maintained by the Unicode Consortium.

**URL**— Universal resource locator. A textual string which provides information used to locate and retrieve information over the Internet.

**UTF-8**— Universal transfer format - 8. A variable-byte encoding of Unicode strings that ensures that no embedded null bytes exist within a string of characters.

**VPN**— Virtual private network. A protected communications path over unsecured networks between two cooperating parties.

**White pages**— A name-based directory. White pages directory lookups are done by name.

**XML**— Extensible markup language. A markup language used to describe arbitrarily complex data. Distinguished by the specification of open-ended custom data type definitions. Becoming important in distributed applications and data-interchange.

**X.500**— Series of directory standards prior to LDAP.

**Yellow pages**— An attribute-based directory. Yellow pages directory lookups are done based on some set of attributes about an entry but not necessarily its name.
Following are some selected sources and brief annotations. Appendix H contains links to many other bibliographical sources, such as IETF RFCs and other documents.


Essential COM. Box, Don. Addison Wesley Longman. Reading, Mass. 1998. ISBN: 0-201-63446-5. The “why” of COM, as well as the “how” is covered by one of the leading authorities on Microsoft's Component Object Model.


OS/390 Security Server LDAP Server Administration and Usage Guide. IBM publication number SC24-5861-04. This document is on the CD and describes how to set up and use the LDAP server on OS/390.

OS/390 LDAP Client Application Development Guide and Reference. IBM publication number SC24-5878-00. This document is on the CD and describes the LDAP C API available for use on OS/390 systems.


Understanding X.500—The Directory. Chadwick, David W. Chapman and Hall, Salford, UK. 1994. ISBN: 0-412-43020-7. This book is no longer in print, but is available on-line (at the time of this writing) at http://www.salford.ac.uk/its024/Version.Web/. This is a very good introduction to the historical roots of LDAP and to directory issues in general.


X.500 Series Recommendations. ITU. 1993. This family of documents defines the X.500 architecture, schema, and protocols.
A
Abstract object classes, 187, 204
Access control, 467
Accountability, 125
  risks, 127
  threats, 126
ACEs (access control entries), 220, 425–426
ACID (atomicity, consistency, isolation, and
durability), 16, 148, 467
ACL (access control list), 152, 220, 467
aCorporateSchema package, 203
Active Directory, 47, 67–68, 270–271, 351
  access control model, 220
  ACLs (access control lists), 220
  DNS services, 45
  Kerberos, 129
Active Directory access control model, 424–427
  ACL (access control list) structure, 424–426
  allowing modification, 426
  evaluation algorithm for access check, 426
  granting anonymous access, 426
  object creation, 426
  preventing access, 426–427
  problems and solutions, 426
  users and groups in ACLs (access control lists),
  424
ActiveX, 467
ActiveX controls, 402–403
Add operation, 374
Administrative interoperability, 152
AdsGetObject function, 395
ADSI (Active Directory Services Interface), 256,
  267, 374, 467
C/C++, 395
Microsoft Platform SDK, 396
provider layer, 393
Visual Basic, 395
Advertisers, 40
Advertising services, 103–105
Agents, 6
aGroup object, 209
Algorithm and scalability, 112–113
Allow_object ACE type, 220
Apache Web servers, 344
  administration of, 348
  logs, 349–350
  Mod Perl library, 347
aPerson object class, 209
APIs (application programming interfaces),
  253–254, 467
APPLET tag, 401–402
Application directories, 26–27, 467
Application management
  directory, 323–324
  directory-enabling client, 326–327
  directory-enabling server, 324–326
  problem, 321
  purpose of, 322–323
Application model for enterprise software, 85–90
Application object, 282
Applications, 81–84
  administration, 179–180
  availability, 145
  client-server, 86
  client-side caching, 123
  coding errors, 128
  configuration, 81–82
  configuring, 6
  consistent security model, 40
  consistent user interface, 34–35
Applications, continued
  deployment, 6
  development and maintenance as source of risk, 127–128
  directory usage, 20–21
  distributed administration, 34
  extensibility, 179
  external attacks, 127
  future demands, 34
  generic directory service, 34
  identifying, 8
  improving availability, 102–106
  inside development attacks, 127
  installing, 6
  intentional internal attacks, 127
  intra-company, 108
  levels-of-recovery scheme, 99
  maintenance, 6
  monolithic, 85–86
  navigation pane, 34–35
  network access addresses and protocols, 82
  n-tier, 90
  operating system logons, 335
  password caching, 335
  performance, 117, 177–178
  planning ahead for scalability, 109
  policy-based administration, 34
  poor software design or coding, 127
  reliability, 69
  scalability, 107–109
  scalability envelope, 8–9
  single sign-on, 335–336
  skills portability, 35
  small-office, 107
  SMP (symmetric multiple processors), 93–94
  style data, 60
  subject pane, 34
  synchronization of updates, 177–178
  thin client, 87–88
  three-tier, 89
  tiered, 88
  two-tier, 89
  unintended weakness, 127
  unique features, 197–198
  user and group definitions, 83–84
  Web farm, 340
  workgroups, 107–108
  Application server, 467
  Application-specific directories, 46
  ASCII (American Standard Character Information Interchange), 467
  ASN.1 (Abstract Syntax Notation 1), 42, 194–195, 467
  ASP (Active Server Pages), 347, 409–410, 468
  Atomicity characteristics, 209
  Attributes, 372–373
  Attributes class, 387–388
  Attribute types, 180, 184–186, 202, 451–460, 468
  corporate employee directory, 296
  defining, 206–208
  definition, 185, 202
  distinguishedName syntax, 209
  dn-pointers, 209
  dynamic characteristics, 209
  format, 206
  grouping, 186–187, 202
  matching rules, 206–207
  multivalued, 185–186
  name, 184–185
  OID (object identifier), 185, 206–207
  reusing and extending, 202–203
  single-valued, 185
  syntax, 206
  unique identifier, 206
  attributeTypes attribute type, 192, 212
  Auditing, 131
  Audit records, 131–132
  Authentication, 468
  accessing existing backend systems, 129
  dedicated authentication service, 129
  directories, 129
  directory-based, 8
  local security database, 128–129
  operating system logons, 335
  public key certificates, 335
  service ID, 218–219
  user ID and password, 218–219
  user identity mapping, 219–220
  users, 128–130, 217–220
  Web servers, 411
  World Wide Web, 411
  Automatic fail-over, 137–139
  Auto-restart, 137
  Auxiliary object classes, 187, 204
  AVA (attribute value assertion), 468
Availability, 14, 19, 95–97
  applications, 145
  high service availability, 97–100
  improving application availability, 102–106
  low service availability, 100
  quotation application, 330–331
  servers, 96–97
  services, 96–97
  Web farm, 353–354
AWLEnterpriseDirectories object, 164–165

B
Backend systems, 218
Backups and directories, 140–141
Bandwidth, 121
Banyan Vines, 271
baseDn distinguished name, 378
Base schema, reusing, 211
basicAttributes collection, 390
B2B (business-to-business) environments, 75–76, 468
Beans, 402
BER (basic encoding rules), 468
Binary syntax, 181, 183
BIND, 345
bind function, 386
bind() method, 265, 386
Bind operation, 217–218, 374
Boolean syntax, 181–182
Boss thread, 91
Boss-worker thread model, 91
Brooks, Frederick P., 55
Browser-based applications, 111–112
Browsers
  cookies, 407–408
  versions, 463
Bubble sort algorithm, 112
Buffer Management functions, 414–415
Bulk-load scenarios, 117
Business issues impacting structure of custom
  schema, 210–211

C
Caching versus replication, 228–229
Capacity, 118
C API, 9, 20–21, 374
  ld handle, 375
Case-exact matching, 182
  caseExactMatch matching rule, 183
  caseExactOrderingMatch matching rule, 183
  caseIgnoreIA5Match matching rule, 207
  Case-ignore matching, 182
  caseIgnoreMatch matching rule, 183
Case studies
  application management, 321–331
  corporate employee directory, 294–310
  customized Web sites, 311–319
  Internet and intranet single-sign, 333–337
  roaming users with LDAP, 277–293
  Web farm, 339–355
Categorizing data, 163
Cauchoo’s Resin, 411
C/C++ and ADSI, 395
CCITT (International Telegraph and Telephone
  Consultative Committee), 468
CCITT (Consultative Committee on
  International Telephone and Telegraphy), 32
CD-ROM (enclosed with this book)
  browser versions, 463
  contents, 461–462
  executable programs, 463
  executable programs and platforms, 463
  file formats, 463
  program environment, 464–465
  readme.htm pages, 461
  requirements for running samples, 464
  sample programs, 464
  startup.exe program, 463
  welcom.htm file, 463
CDS (Cell Directory Services), 258–259, 468
CDS clients, 40–41
CDS servers, 40–41
CGI (common gateway interface), 88, 404–405, 468
CGI applications, 347
Chaining, 238
Change latency, 19
CICS (Customer Information and Control
  System), 468
CIM (Common Information Model), 80, 198, 468
Classes, 150
Clerks, 40
Client program, 103–105
Clients, 468
  client-server paradigm, 144
  directory-enabling, 326–327
Clients, continued
finding servers, 103–105
life cycle control changes, 144–145
offloading processing to, 110–112
projecting static brochures to, 397–398
thick, 110–111, 144
thin, 110–111
Client-server, 468
Client-server applications, 86
Client-server glue technologies, 90–92
managed components, 92
MOM (message-oriented middleware), 90–91
Client-server paradigm, 144
Client-side APIs
ActiveX controls, 402–403
DHTML (dynamic HTML) with CSS (Cascading Style Sheets), 403–404
Java applets, 401–402
Client-side caching, 123
Client-side scripting, 400–401
Clustering, 72, 93, 469
fail-over clustering, 137–138
servers, 114
Code or application isolation, 128
Code quality, 128
Coding errors, 128
Collaborative systems, 109
COM+ (Enhanced Microsoft Component Object Model), 469
COM (Microsoft Component Object Model), 469
COM components, 409
COM+ components, 409
Common directory programming interfaces, 261–267
Common directory protocols, 259–261
Communications of the ACM, 57
Compare operation, 374
Component instance manager, 92
Config.sys file, 78
Consistent networking protocol, 40
ConsoleOne, 270
Console window, 61
Constraints, 120
Consumer software, 56
Contact object, 282
Containers, 169
Containment relationship, 201
Context interface, 265–266, 383–384, 386, 390
Cookies, 335, 407–408
CORBA (common object request broker architecture), 469
CORBA with IIOP, 335, 400
Corporate directories, 33, 469
use cases, 294
Corporate employee directory accessing, 299–303
adding distribution list, 301–302
adding information, 304
attribute types, 296
company reorganization, 308
data model, 295–298
deleting distribution list, 302
deleting information, 304
e-mail systems, 309
employee transfers, 305–308
issues not addressed, 309–310
looking up chain of command, 300–301
looking up people, 299
managing information, 303–308
modifying distribution list, 302–303
modifying information, 304
name space layout, 297–298
object classes, 295
purpose of, 293–294
sample code, 308–309
COSNaming (Common Object Services Naming), 256, 265, 383, 469
COSNaming over IIOP, 260–261
Cost
categories, 57
complexity and, 55
deployment, 57
development, 57–58
and enterprise characteristics, 54
of failure, 57–58
life-cycle, 57–58
maintenance, 57
per line of code, 55
createSubcontext() method, 390
Credential management functions, 416–417
CSS (Cascading Style Sheets), 403
Customer object, 282
Customized Web sites, 311–319
Custom schemas, 197–198
attribute types definition, 206–208
business issues impacting structure of, 210–211
DACL (discretionary access-control list), 425
Daemon, 469
DAP (directory access protocol), 255, 469
Data, 5
ACID (atomicity, consistency, isolation, and durability), 147
categorizing, 163
change latency, 19
confidentiality, 125–127, 222–223
consistent state, 233
cross-platform, 147
dynamic partitioning, 150
easy partitioning, 150
entering and using, 163
enterprise software storage, 161–163
getting out of directory, 20–21
high RASSS, 147
identity, 8
impedance mismatch, 15
indexing, 122–123
inefficient and ineffective protection, 125
integrity, 125–127, 222–223
kinds belonging to directory, 18–20
loose consistency, 17–18
management, 168–170
multivalued, 147
operations, 168–170
organizing for performance, 161–162
partitioning, 233–238
rate of change, 15
replication, 226
requirements, 147
search and ordering capabilities, 147
streaming to file, 143
structuring, 162
white pages retrieval, 164–166
yellow pages retrieval, 166
Databases
ACID properties, 16–18
system resources, 58
Database server, 86
Data communications protocols, 253
Data model
corporate employee directory, 295–298
one-to-many relationship, 200
Data retrieval, 163–168
Data sharing, 4–5
DBMS (database management systems), 86, 346, 469
DCE (Distributed Computing Environment), 258–259, 469
advertisers, 40
CDS clients, 40–41
CDS servers, 40–41
C language programming interfaces, 258–259
clerks, 40
RPC (remote procedure call), 258
DCE CDS, 254
DCOM (distributed component object model), 335, 399–400, 469
DDNS (dynamic DNS), 469
Defensive coding, 128
Defining
attribute types, 206–208
functional characteristics, 209–210
object classes, 204–206
packages, 202–203
DEN (Directory Enabled Networking), 198, 469
Denial of service attack, 168
deny_object ACE type, 220
Deployment cost, 57
Description, 209
Development cost, 57–58
Devices, managing, 154–155
/dev portion of file system, 78
Dewey decimal system, 25–26
DFS (distributed file service), 41
DHTML (dynamic HTML) with CSS
(Cascading Style Sheets), 403–404
DIF (Directory Interoperability Forum), 248
Directories
  access control problems, 246
  active manipulation, 4
  agents, 6
  all-or-nothing administrative rights, 246
  applications, 26–27, 81–84
  attributes, 372–373
  authentication, 129
  backups, 140–140
  better multivendor interoperability, 151–152
  categorizing data, 163
  as central access control mechanism, 129–130
  computer identity, 15
  containers, 169
  controlling access to entries, 220–222
  controlling format of data, 173
  corporate, 33
  data, 5
  data sharing, 4–5
  definition of, 10
  distinguished, 373
  distributed, 17
  DNS (Domain Name Service), 37, 75
  dynamic partitioning, 150
  early standardization, 37–43
  easy partitioning, 150
  electronic, 3–4
  enabling roaming, 277–278
  entering and using data, 163
  enterprise data storage, 161
  entry, 469
  exploitation, 58–59
  extra names, 373
  future enhancements for, 148–155
  generic, 32–35
  getting data out of, 20–21
  granularity, 246
  hardware configuration, 78–80
  hierarchical model, 169
  hierarchical view, 58
  hierarchy, 469
  HR (human resource) information, 33–34
  HTML-based client upgrade, 144–145
  improved administration, 153
  integration with operating system, 148
  isolation, 16–17
  lack of access control standardization, 220
  locating remote computers and software, 82
  locator services, 169
  maintaining operating systems from, 7
  managing, 160–161
  monitoring tools, 153
  networks, 31–32, 80–81
  operating-system-neutral, 7
  operating systems, 29–30, 77–80
  operation of, 15
  organizing data for performance, 161–162
  people, 75–77
  personal, 27–29
  phone book, 24–26
  read-mostly data, 15
  redundancy, 139–140
  referrals, 152, 236–237
  relative distinguished name, 373
  scalability, 115
  as service locator service, 5
  single sign-on, 336–337
  size, 336
  SNA (system network architecture), 31
  SNA Host Names and Logical Units, 37
  standards-based, 7
  storing identity and credential mappings, 336
  storing large objects, 150
  storing public key certificates, 336
  structure and TSA (The Sales Application), 286–287
  structuring data, 162, 373
  system, 23–24
  system information, 5
  system view, 5–9
  today's standards, 43–48
  types, 23–27
  user-oriented, 3–5, 27–32
  users, 75–84
  vendor differences and TSA (The Sales Application), 285–286
  Directory-based authentication, 8
  Directory data model, 372–374
  Directory-enabled server, 142
  Directory information tree, 470
  Directory programming interfaces, 261–267
  Directory protocols, 257–261
  Directory servers, 13, 470
  Directory service browser, 179
Directory services, 4, 7, 10, 13, 18, 470
additions and changes, 177
aspects of, 173–174
as authentication service, 336
availability, 14, 19, 99
characteristics, 58
commercial implementations, 269–271
deep hierarchy, 211
defining information content, 206
direct access to information, 178
distinguished name, 192
dynamic redirection, 68
ease of use, 173
fail-over support, 70–73
fault tolerance, 70–73
function characteristics, 16
generic, 32–35
impedance mismatch, 15
kinds of data belonging to, 18–20
loose consistency, 17–18
managing, 159–170
open source implementation, 271–272
out-of-the-box, 67
performance, 173
political dimension, 159–160
publishing and locating services for clients, 346
RASSS characteristics, 14–15
redundancy, 68, 70–73
reliability, 68–69
replication, 15, 226–227
role of, 18–20
scalability, 14–15
sending and receiving messages, 9
shallow hierarchy of entries, 211
specialized implementations, 272
storing strings, 182
synchronization, 241
trees, 242
usage rating, 68–69
users, 69
working description, 21
Directory Services Markup Language, 444
directory synchronization
See synchronization
DirEntry C++ class instance, 378
Distinguished, 373
distinguishedNameMatch matching rule, 209
Distributed administration, 34
Distributed applications and networking
protocols, 40
Distributed directory, 17
Distributed transaction controller, 470
Distribution lists, 28
DIT (Directory Information Table)
name space layout, 193
TSA (The Sales Application), 283
DIT (Directory Information Tree), 180, 189
DIT structure rules, 192, 470
DIT superiors, 205
DMTF (Distributed Management Task Force),
80, 154, 198, 470
DN (distinguished name), 192, 207, 470
dn-pointers, 178–179, 199, 209, 470
DNS (Domain Name Service), 44–46, 260, 272, 346, 383, 397, 470
as generic directory service, 38
host name-to-IP address lookup services, 39
mapping to correct IP addresses, 349
records, 31
referral mechanism, 38
resolver, 31
top layer of name space, 38
DNS servers
BIND (Berkeley Internet Name Daemon)
protocol, 45
built on existing directory service
implementations, 45
communications protocol, 38
implementations, 44
resolvers, 45
SRV records, 39–40
Unix-based operating environments, 44–45
DNS services, 45
Domino, 470
Don’t care directory exploitation, 59
Don’t care software, 53–56, 59
Downtime
controlling, 134–139
failing request versus failing server application,
136–139
real-time event logging with first failure data
capture, 134–136
DSAs (directory service agents), 259, 470
DSML (directory service markup language), 153, 470
schema, 215
DSP (directory access protocol), 42
DUAs (directory user agents), 259, 470
Dynamic DNS servers, 39
Dynamic load-balancing server, 71
Dynamic partitioning, 150
Dynamic replica creation, 151

E
Ease of use, 173
Easy partitioning, 150
EBCDIC (extended binary coded decimal interchange code), 470
ECC (error correcting code), 70–71, 471
EcmaScript, 409
EJB (Enterprise Java Beans), 409, 471
EJS (enterprise Java server), 471
Electronic directories, 3–4, 75–76
Employee directory, 3–5
Encina-based transaction processing, 41
Enterprise, 471
Enterprise application models
client-server applications, 86
monolithic applications, 85–86
thick client applications, 86–87
thin client applications, 87–88
tiered applications, 88–90
Enterprise data operations and management, 160–161
Enterprise data retrieval, 160
Enterprise data storage, 160
Enterprise directory, 471
Enterprise-rated directory exploitation, 59
Enterprise-rated software, 53–56, 59–60
Enterprise software, 471
availability, 95–96
characteristics, 8–9
data operations, 160, 168–170
data retrieval, 160, 163–168
data storage, 160, 161–163
locator services, 169
management, 160–161
performance bottlenecks, 165
RASSS (reliability, availability, serviceability, scalability, and security), 14
respecting characteristics, 8
scalability, 107
security, 8
Entry, 471
Equality matching, 183
Equality matching rule, 206–207
E-R (entity-relationship) model, 199–200
Error returns, 60
ETS (Enterprise Transaction Server), 411
Event log requestor, 136
Executable programs, 463
Existing system directories, 46
External attacks, 127

F
Failing
cost of, 57–58
data capture of first, 134–136
requests, 136–139
server application, 136–139
Fail-over clusters, 93–94, 137–138
Fail-over support, 70–73
Fault tolerance, 70–73
Federal Reserve Bank, 19–20
FGI (Fast CGI), 406
File formats, 463
Files
flat text, 3
formats, 3
streaming data to, 143
File system management, 78
Flat text files, 3
Fluctuations, 121–122
Framework, 471
FTP, 399
Future enhancements for directories, 148–155

G
generalizedTime syntax, 181–183
Generic directories, 32–35, 82
Generic directory services, 32–35, 47, 471
consistent user interface, 35
extensibility, 33
storing and retrieving information, 83–84
Generic distributed directory service, 46
getAttributes() method, 387
gethostbyname() function, 45–46, 260
GNUJSP, 411
Good-enough directory exploitation, 59
Good-enough software, 53–56, 59
Gopher, 399
Graceful fail-over with takeover, 72
Grouping attribute types, 186–187
  groupNames attribute type, 209
Groups, 77
  adding users, 209
  mail systems, 28
GSS-API (Generic Security Services), 413–420, 471
  Buffer Management functions, 414–415
  context management functions, 417–419
  converting numeric status codes into printable messages, 416
  credential management functions, 416–417
  general helper functions, 414–416
  Mechanism/Package Management functions, 414
  message protection functions, 420
  name management functions, 415
  object identifier helper functions, 415–416
GUID (globally unique identifier), 471
GUI responsiveness, 61

H
Hardware configuration, 78–80
Hardware servers, 96
Heap sort algorithm, 112
Heterogeneous distributed computing, 7–8
Home-environment applications scalability, 107
Host names, 37
HR (human resource) information, 33–34
HTML (hypertext markup language), 398, 471
  HTML-based client upgrade for directories, 144–145
HTTP (hypertext transport protocol), 397, 399, 471
HTTP cookies, 342

I
IA5, 471
  IADsContainer interface, 395
IADS interface, 395
  IADSSOpenDSObject interface, 395
IANA (Internet Access Naming Authority), 471
IANA (Internet Address Naming Authority), 38
IBM schema, 211
IBM SecureWay Directory, 33, 212, 220, 269
  access control model, 321–324
  ACL (access control list) structure, 422–423
  allowing modification, 423
  client public key certificates, 129
  evaluation algorithm for access check, 423
  granting anonymous access, 423
  object creation, 423
  preventing access, 424
  problems and solutions, 423–424
  users and groups in ACLs (access control lists), 421–422
IBM standard schema
  attributes, 432
  binary syntax, 432–433
  boolean syntax, 433
  GeneralizedTime syntax, 433
  standard DIT (directory information tree), 431
  standard objects, 431
  syntaxes, 432–433
telephoneNumber syntax, 433
IBM TxsSeries, 409
IBM Websphere Enterprise Edition, 409
idapmodify utility, 352
Identity
  data, 8
  managing, 154
  users, 217–220
  IDirectoryObject interface, 395
  IDirectorySearch interface, 395
  IDL (interface definition language), 254, 471
IEEE (Institute of Electrical and Electronics Engineers), 471
IETF (Internet Engineering Task Force), 249, 273, 472
IETF documents, 443
IETF drafts, 440–443
IETF RFC 883, 260
IETF RFC 1777, 46
IETF RFC 1823, 20
IETF RFC 2252, 181, 183–184, 186, 211
IETF RFC 2535, 260
IETF RFCs, 38, 435–440
IIOP (Internet Inter-ORB Protocol), 260, 472
IIS (Internet Information Server), 410
Indexing data, 122–123
Inetorgperson objects, 262
Information performance and usability, 200
Information services, 6–7
InitialDirContext class, 385
Input/output usage, 120
Inside development attacks, 127
IntegerMatch matching rule, 183
Integer syntax, 181, 182
Intel-based Linux, 344
Intentional internal attacks, 127
Internet
   identity, 154
   users, 333–334
Internet and intranet single-sign, 333–337
Interoperability, 151–152
Intra-enterprise applications, 109
Intranet users, 333–334
IODF (I/O definition file), 78
IP addresses, 31
IPX/SP, 472
Iron triangle, 58
ISAPI, 406
Isolation, 16–17, 96
ITU-T (International Telecommunication Union-Telecommunication Standardization Bureau), 472

J
Java, 401
   JIT (just-in-time) compiler, 406
Java applets, 400–402
Java beans, 402
java.naming.factory.initial property, 385
java.naming.provider.url property, 385
JavaScript, 409
Java servlets, 406
JNDI (Java Naming and Directory Interface), 254, 256, 265–267, 374, 383, 444, 472
   accessing LDAP directories, 383–390
   add operation, 389–390
   compare operation, 387–389
   delete operation, 389–390
   getting LDAP DirContext, 385
   initialization and termination, 384–385
   modify name operation, 390
   modify operation, 389–390
   results, 385–386
   search operation, 387–389
JNDI bind, 386–384
JScript, 409
JSP (Java server pages), 410–411
JSWDK, 411
JTA/JTS (Java transaction API, Java transaction service), 472

K
Kerberos, 40, 129, 472
   BIND, 217
   single sign-on, 336

L
ladp_modify() function, 222
ladp_search_s() function, 378
ladp_bind() function, 377
ladp_compare_s() function, 378
LAN managers, 41
Latency, 121–122, 228
   as access protocol, 46
   add operation, 374, 379–380
   application-specific directories, 46
   attributes, 373
   attribute types, 184–186
   bind operation, 374
   as C API, 9
   C API, 20–21, 374
   compare operation, 374, 377–379
   delete operation, 374, 379
   directory data model, 372–374
   enterprise data retrieval, 163–168
   existing system directories, 46
   getting LDAP handle, 375–376
   initialization and termination, 374–375
   interoperability, 10
   LDAPConnection C++ class, 381
   ld handle, 374–376
   modify name operation, 374, 381
   modify operation, 374, 380
   operations, 372
   as protocol, 9
   results messages, 376
   roaming users with, 277–293
   schema, 180–192, 181
   search operation, 374, 377–379
   search referral messages, 372
SSL (secured sockets layer), 222
subschema object class, 212
support to X.500 directories, 47
syntaxes, 181–183
Version 3 updating schema, 212–213
white pages data retrieval, 164–166
X.tpp directory model, 372
yellow pages data retrieval, 166–168
ldap_add() function, 379
ldap_add_s() function, 379–380
ldapadd utility, 352
LDAP API BIND operations, 217–218
ldapasc program, 121
LDAP-assessible directory servers, 383
ldapauth program, 130
LDAP_AUTH_SIMPLE bind type, 377
ldap_bind() function, 222
ldap_bind_s() function, 217–218
LDAP bind operation, 377
ldap_bind_s() function, 377
LDAP C language API, 254
LDAP C language programming interface, 255, 261–262
LDAP client utilities, 352
LDAPConnection C++ class, 378, 381
LDAPConnection Java interface, 262
LDAPCtxFactory class, 385
LDAPD, 272
ldap_delete_s() function, 379
ldapdelete utility, 352
LDAP directories, 67
LDAP directory server and Web farm, 344–345
ldapdpasswd utility, 352
ldapdelete utility, 352
LDAP directories, 67
LDAP directory server and Web farm, 344–345
ldapdelete utility, 352
ldap.h header file, 375
ldapindexperf program, 123
ldap_init() function, 375
ldap_kerberos_bind_s() function, 217
ldapmodify, 212
ldap_modify_s() function, 380
LDAPMod structure, 380
ldap_open() function, 375
ldap_parse_result() function, 376
ldap_result() function, 376, 377
ldap_sasl_bind_s() function, 217–218
ldap_search API, 164
ldap_search() function, 221–222
ldap_search_s() API, 20–21
LDAP servers, 472
communications with, 59–60
default referral, 47
locating, 61–64
referral objects, 47
ldap_simple_bind_s() function, 217–218
ldapsyn program, 121
ldap_unbind() function, 376
LDAP V2 schema, 180
LDAP V3 schema, 180
ld handle, 375
C API, 375
getting, 375–376
LDIF (LDAP Data Interchange Format), 141, 207, 212–214, 472
LDUP (LDAP Duplication/Replication/Update Protocols), 248
Legacy server, 324
Library card catalog, 25–26
Life cycle control, 133
applying, 143–145
automatic fail-over, 137–139
auto-restart, 137
client and server changes, 144–145
directory redundancy, 139–140
installing new server in place, 142
periodic maintenance, 140–141
streaming data to file, 143
Linux, 344
DBMSs, 346
DB2 UDB, 350
reliability, 353
Linux distributions
BIND, 345
OpenLDAP, 67
PostgreSQL, 346
slapd, 345
slurpd, 345
Load balancing, 93
Local cache, 60
Local security database, 128–129
locatew program, 61–64, 104
Location services, 102–103
Locator services, 60, 169
Logging real-time events, 134–136
Logical data model, 200
Loops and referrals, 237
Loose consistency, 17–18
Lotus Domino Name and Address Book, 271
Lotus Notes, 109, 222
LUs (logical units), 37, 472

M
Mail systems, 27–29
Maintainability and software life cycle, 133–134
Managed components, 92
Managing
data, 168
directories, 160–161
directory services, 159–170
information services, 6–7
Mandatory information, 176
Man-rated directory exploitation, 59
Man-rated software, 20, 53–56, 59–61
Manual synchronization, 244
Matching rules, 183–184, 206–207, 472
MDS, 351
Mechanism/Package Management functions, 414
Message protection functions, 420
Metadirectories, 30, 33, 241, 249–250, 472
Metadirectory services, 30
Microsoft Platform SDK, 396
Microsoft Visual Studio, 410
Middle-tier applications and Web farms, 346–347
MIME (multipurpose Internet mail extensions), 400
ModificationItem objects, 390
modifyAttributes() method, 390
Modify name operation, 374
Modify operation, 374
Mod Perl library, 347
Mod Pod scalability, 354
MOM (message-oriented middleware), 90–91, 472
Monolithic applications, 85–86
MQSeries, 472
MSDN (Microsoft Developer Network), 396
MSMQ (Microsoft Message Queuing), 472
MSNBC Web farm, 339
MTA (Mail Transfer Agent), 473
MTS (Microsoft Transaction Server), 409
Multimaster replications, 140, 227, 231
Multiple-system scaling, 114
Multiple threading errors, 120
Multiprocessor systems, 112
Multiprocess software services, 96
Multithreaded programs, 118–119
Multithreading, 112
Multiuser operating system, 23
Multivalued attributes, 373
Multivalued attribute type, 185–186
Multivendor interoperability, 151–152
(The) Mythical Man Month, 55

N
Name management functions, 415
Name space, 473
corporate employee directory layout, 297–298
efficient retrieval of department, 201–202
exploiting hierarchy, 200–202
extracting more detailed information, 200–201
grouping related information, 200
implementing containment relationship, 201
implementing relationships, 200
layout, 193
location of package definitions, 203
naming entries, 205
Naming
attribute types, 205
constraints, 192–193, 473
NamingEnumeration class, 388
NamingException exception, 385–386
Navigation pane, 34–35
NCPs (Network Core Protocols), 257–258
NCPs (Novell Core Protocols), 473
NDAP (Novell Directory Access Protocol), 260, 473
NDK (Novell Development Kit), 263–264
NDS (Network Directory Service), 47
NDS (Novell Directory Service), 33, 257–258, 269–270
Netscape Directory, 220
Netscape Directory access control model
allowing modification, 429
bind rules, 427–428
composed ACI, 428
evaluation algorithm for access check, 429
granting anonymous access, 429
permissions, 427
preventing access, 429
problems and solutions, 429
structure of ACL (access control list), 427–428
target information, 427
users and groups in ACLs (access control lists), 427
Netscape Directory Server, 270
Netscape Directory Services, 33
NetVision Synchronicity, 270
Netware
  DNS services, 45
  identity, 154
Network bandwidth, 121–122
Networking protocols, 40
Networks, 80–81
  bandwidth, 121
  directories, 31–32
  latency, 122
  policy-based administration, 80
  timing variation, 122
  usage and performance, 121–122
  VPNs (virtual private networks), 80
NLM (NetWare Loadable Module), 270
NNTP (network news transfer protocol), 399
Nondirectory object, 220–222
Objects, 150, 473
N-tier, 473
N-tier architecture and servers, 114
N-tier Web applications
NTLM (Windows NT LAN Manager), 411
NWDSSearch API, 264

O
Oak, 401
objectClass attribute, 373
Object classes, 186–187, 202, 447–450, 473
  attribute types, 204–205
  corporate employee directory, 295
  data modeling constraints, 206
  defining, 188, 204–206
  definitions, 202
  DIT superiors, 205
  dynamic characteristics, 209
  format of entries, 205
  grouping, 202, 206
  names, 187
  OID setting, 206
  optional attribute types, 205
  required attribute types, 204–205
  reusing, 205
  reusing and extending, 202–203
  unique identifier, 206
  objectClasses attribute type, 192, 212
  Object class hierarchy, 188, 473
  Objects, 150, 473
    creation and use, 163
    proxy, 220
    storing large, 150
  OBJECT tag, 401–402, 402
  OcX (OLE control), 402
  Offloading processing, 114
  OID (Object Identifier), 183, 206, 473
    helper functions, 415–416
  OMG (Object Management Group), 383, 473
    Object Management Group) COSNaming
      interface, 254, 261
    bind function, 386
  OMG (Object Management Group) Name
    Service, 82
OpenLDAP, 67, 351
OpenLDAP Foundation, 272
OpenLDAP organization, 248, 345
  operating-system-neutral, 7
Operating systems
  agents, 6
  directories, 29–30, 77–80
  directory integration, 148
  fail-over clustering, 138
  groups, 77
  hardware configuration, 78–80
  hierarchy of isolation, 98
  identify, 8
  logons and authentication, 335
  maintaining from directories, 7
  managing file systems, 78
  multiuser, 23
  robust behavior, 98
  roles, 67–68, 77
  services, 71
  usage rating, 68
  use of generic directory services, 78
  user definitions, 77
  users, 77–80
  users and group information, 29–30
Optional attribute types, 205
Optional information, 176
ORD (object request broker), 265
Ordered matching, 184
organizationalUnit object class, 192–193
Organization object class, 192–193
OSF (Open Software Foundation), 258
OSF DCE RPC, 335
OSI (open systems interconnection), 33, 42, 473
OS/390 operating system, 98–99
OS/390 Security Server: LDAP Client Application Development Guide and Reference, 21
OU (organization unit), 21
Out-of-the-box directory services, 67

P
Packages, 206–207
defining, 202–203
description and version, 202
PARAMETER tag, 401
Parent object class, 188, 205
Partitioning, 150, 225, 473
application-level problems, 235–236
business issues, 239
management, 235
performance, 234–235
referrals, 236–237
replication and, 238
search performance, 236
Partitions, locating correct, 235–236
Password caching, 335
Passwords
synchronization, 335
users, 30
People, 75–77
Performance, 96, 117–118, 173
adverse affect by referrals, 237
bottlenecks, 165
constraints, 120
fluctuations, 121–122
gating factors, 118–122
input/output usage, 120
latency, 121–122
measurement, 117–118
monitoring, 153, 349
network bandwidth, 121–122
network usage and, 121–122
partitioning, 234–235
poor, 122–123
processor usage, 118–120
response time, 117–118
schema, 177–178
throughput, 117
timing, 121–122
Periodic maintenance, 140–141
Perl, 347
Personal directories, 27–30
Person object class, 197–198
Phone book directories, 24–26
Physical data model, 200, 202
Planning ahead for scalability, 109
PMW (Piscataway Machine Works)
customer information, 361
directory architecture, 360–363
employee information, 361
to enterprise directory service and migration, 365
Internet technology, 358–359
metadirectory, 364
observations and recommendations, 363–364
ownership/access permission, 365
synchronization, 365–366
Policy-based administration, 34
PostgreSQL server logs, 349–350
Privacy, 125–127
Processes, 112
Processor usage, 118–120
Product object, 282
Programming interfaces, 253–259
Projecting static brochures to clients, 397–398
Property, 473
Protecting object, 220
Protocols, 474
protocols, 253–255
Web, 399–400
Proxy objects, 220, 221
Public key certificates, 335
Public key support
on, 336
Publishing schema, 190–192

Q
QoS (quality of service), 154
Quotation application, 326
availability, 330–331
critique, 328–329
reliability, 330
scalability, 331
schema, 327
security, 331
serviceability, 331

R
RADIUS (remote authentication dial-in user service), 474
RADIUS servers, 80
RAID (redundant array of inexpensive devices), 474
RASSS (reliability, availability, serviceability, scalability, and security), 14, 474
RDN (relative distinguished name), 192, 474
Real-time event logging with first failure data capture, 134–136
Redundancy, 70–73, 96
Redundant computer systems, 70
Redundant directories, 139–140
Redundant servers, 99–100
Referential integrity and schema, 215
Referrals, 152, 236–238
Refetch service, 230
Regular BIND, 217
Reliability, 95
    quotation application, 330
    Web farm, 353
rename() method, 390
Replicas, 73, 151
Replication, 225, 241–243, 245, 474
    application-level problems, 228
    business issues, 239
    versus caching, 228–229
    collisions, 231–233
    latency issues, 228
    loose consistency, 228
    multimaster, 227
    operation of, 226–227
    partitioning and, 238
    problems, 247–248
    single master, 227
    synchronization agents, 226
    temporal consistency, 228–233
Requests, failing, 136–139
Required attribute types, 204–205
resolve() method, 265
Resolvers, 31, 45, 260
Resolver service, 45–46

Resource monitor, 474
Response time, 117–118
Retrieving data, 163–168
Retry program, 229–231
Reusing base schema, 211
RFC 2165, 273
RFC (request for comments), 474
Risk analysis, 126
Risks, 127–128
RMI (remote method invocation), 400, 474
RMI/IIOP, 474
RMI over IIOP, 400
Roaming
    enabling, 277–278
    problem description, 278–279
    users with LDAP, 277–293
Robust server code, 97–99
RocketJSP, 411
Roles, 77
RPC (remote procedure call), 40, 254, 474

S
SACL (system access-control list), 425
SASL (Simple Authentication and Security Layer), 474
SASL (Simple Authentication and Security Layer)
        BIND, 217
scalability, 14–15, 96
    algorithm choice, 112–113
    approaches, 110
    directories, 115
    home-environment applications, 107
    intra-company applications, 108
    multiple-system scaling, 114
    multiprocessor systems, 112
    multithreading, 112
    offloading processing to client, 110–112
    planning ahead for, 109
    quotation application, 331
    single-system scaling, 110–113
    small-office applications, 107
    Web farm, 354
    workgroup applications, 107–108
schema package, 474
Schemas, 474
    base, 197
    checking, 187–188
Schemas, continued
    class objects and TSA (The Sales Application), 281–282
    common, 197
    custom, 197–198
    defining, 188, 192
    defining information location, 176
    definition of, 180
    describing information, 174–175
    directory entries and attributes, 373
    as directory road map, 174
    DIT structure rules, 192
    DSML (directory service markup language), 215
    extensibility, 179
    function of, 174–176
    LDAP, 180–192
    LDAP Version 3 update, 212–213
    LDIF format update, 212–214
    managing, 202
    mandatory information, 176
    mapping problems, 246
    multiple, 197
    naming constraints, 192–193
    optional information, 176
    performance, 177–178
    publication, 190–192
    quotation application, 327
    referential integrity, 215
    role of, 177–180
    service-administration-point object class, 327
    special process, 209–210
    TSA (The Sales Application), 281–286
    TSA (The Sales Application) attributes, 282–283
    TSA (The Sales Application) issues, 286
    usability, 178–179
    Web farms, 351–352
    WSH (Web Self-Help) site, 312–313
    X.500 data model, 180–181
    XML (extensible markup language), 215
    ScheMax utility, 270
    SCP (serviceConnectionPoint)
        serviceBindingInformation attribute, 105
    Scripts, 474
    search() method, 387
    Search operation, 374
    SearchResult class, 388

Secureway Directory Server, 351
Security, 8
    accountability, 125
    application development and maintenance risks, 127–128
    auditing, 131
    code or application isolation, 128
    code quality, 128
    consistent model, 40
    controlling access by organizational responsibilities, 128–130
    controlling access to directory entries, 220–222
    data confidentiality, 125, 222–223
    data integrity, 125, 222–223
    defensive coding, 128
    enterprise software, 8
    heterogeneous computing, 8
    Java applets, 402
    management and administration, 223
    monitoring, 153
    peer reviewing, 127
    privacy, 125
    quotation application, 331
    risk analysis, 126
    risks, 127
    security policy, 125–126
    threats, 126
    user identification and authentication, 217–220
    Web farm, 354
    Web servers, 411
    Security policy, 125–126
    serializationerror program, 119
    Server application failing, 136–139
    Server-based computing, 111–112
    Server program, 103, 104–105
    Server replicas, 60
    Servers, 13, 96–97, 474
        adding to Web farm, 343–344
        automatic fail-over, 137–139
        auto-restart, 137
        availability, 96–97
        bug or error condition, 100–101
        clustering, 114
        default LDAP port communication, 59–60
        directory-enabling, 324–326
        installing new in place, 142–143
        isolation, 96
legacy, 324
life cycle control changes, 144–145
memory leak, 101
multimaster scheme, 100
n-tier architecture, 114
planning for next release, 141–143
redundancy, 96
redundant, 99–100
reliability, 93
relying on, 60
robust code, 97–99
single-master method, 100
threads, 91
three-tier architecture, 114
upgrading, 141–142
Server-side components, 408–411
Serviceability, 97
  quotation application, 331
  software life cycle, 133–134
  Web farm, 354
  serviceBindingInformation attribute, 105
  serviceConnectionPoint class, 166
  serviceConnectionPoint objects, 164, 167
Service locator service, 5
Services, 13
  advertising, 103–105
  availability, 19
  finding, 103–105
  low availability, 100–102
  operating system, 71
  other uses for, 105–106
  performance, 96
  reasons for creation of, 96–97
  redundant servers, 99–100
  robust server code, 97–99
  scalability, 96
  serviceability, 97
Service window, 95
Servlets, 406
SGML (standard generalized markup language), 474
Shadow objects, 221
Simple BIND, 217
Simple interoperability, 152
Single master replication, 227
Single sign-on, 333
directories, 336–337
Internet and intranet users, 333–334
Kerberos, 336
  public key support, 336
  solution, 335–336
Single-system scaling, 110–113
Single-valued attributes, 373
Single-valued attribute type, 185
Skills portability, 35
slapd, 248, 271, 345, 351
slapd server and PID (program identifier), 352
SLP (Service Location Protocol), 273, 475
slurpd, 248, 271, 345, 351
Small-office applications scalability, 107
SMP (symmetric multiprocessor), 475
  applications, 93–94
SMP systems, 118
SMTP (simple mail transport protocol), 475
SMTP-based mail systems, 31
SNA (systems network architecture), 475
  conversations, 37
directories, 31
host names, 37
LUs (logical units), 37
SOAP (simple object access protocol), 475
Sockets, 37
Software
  classes of usage, 53–56
  consumer, 56
  controlling downtime, 134–139
  cost, 56–58
  cost per line of code, 55
  development costs, 54, 57–58
don’t care, 53–56
  enterprise-rated, 53–56
  failure cost, 57–58
  good-enough, 53–56
  life cycle, 133
  life cycle cost, 57–58
  maintainability, 133–134
  man-rated, 53–56
  serviceability, 133–134
Software servers, 96
Software services, 96
SQL (structured query language), 475
SRV records, 39–40, 475
SSI (server-side include), 409
SSL (secure sockets layer), 222, 475
SSL for Java, 270
SSPI (System Security Provider Interfaces), 475
  Buffer Management functions, 414–415
  context management functions, 417–419
  credential management functions, 416–417
  general helper functions, 414–416
  Mechanism/Package Management functions, 414
  message protection functions, 420
  sspigsp program, 414
  sspipkg package, 414
  Standards-based directory, 7
  startup.exe program, 463
  Stateless servers, 340
  Storing large objects, 150
  Streaming data to file, 143
  String formats, 182
  Strings, 182
  String syntaxes, 182
  Structural object classes, 187, 204
  Structuring data, 162
  Subject pane, 34
  Subschema object class, 212
  subschemasubentry attribute, 192
  Substring matching, 183–184
  Superior object class, 188, 205
  Surrogate objects, 221
  Synchronization, 243–248, 475
    manual, 244
    passwords, 335
    problems, 247
    schema mapping problems, 246
  Syntax, 181–183, 475
  System directories, 23–24, 475
  System.getProperties() function, 385
  System performance, 117
  System view of directories, 5–9

T
  Tasks, 112
  TCP/IP (transmission control protocol/Internet protocol), 475
  TCP/IP networks
    DNS (Domain Name Service), 37
    DNS (domain name service), 31
    IP addresses, 31
    sockets, 37
  Temporal consistency, 18, 229–233
  Thick clients, 110–111, 144, 475
  Thin clients, 87–88, 110–111, 476
  Threads, 91, 112
  Threats, 126
  Three-tier applications, 476
    Web farms, 346
  Three-tier architecture, 114
  Three-tier Web applications, 89
  Throughput, 117
  Tiered applications, 88–90
  Time service, 18–19
  Timestamps, 182–183
  Timing, 121–122
  Tivoli Web site, 170
  Top class, 188
  Top-level DNS names, 38
  toString() method, 386
  Transactional data, 148–150
  Transaction monitor, 159
  Transactions, 90, 118, 476
    Web servers, 408
  Trees, 242
  TSA (The Sales Application), 280–281
    directory structure, 286–287
    directory vendor differences, 285–286
    DIT (Directory Information Table), 283
    missing features, 291–292
    operation of, 287–290
    running on test machine, 290–291
    schema, 281–286
    schema attributes, 282–283
    schema class objects, 281–282
    schema issues, 286
    updating schema, 285
    write activity, 289
    write collisions, 289–290
  Two-tier applications, 89, 476
  Two-tier Web applications, 89
  TXT record, 476

U
  unbind() method, 265, 389
  Unicode, 476
  Unified identity, 8
  Unintended weakness, 127
  Unix-based operating environments and DNS servers, 44–45
Unusual events, 153
Updating schema and TSA (The Sales Application), 285
URL (uniform resource locator), 109, 397, 398, 476
Usage classes, 53–56
directory exploitation, 58–59
User definitions, 77, 217
User IDs, 30
User interface consistency, 34–35
User object, 281–282
User-oriented directories
mail systems, 27–29
personal directory, 27–29
user-oriented directories
view of, 3–5
Users, 75–84
adding to groups, 209
applications, 81–84
authentication, 128–130, 217–220
controlling access by responsibilities, 128–130
entering and using data, 163
identification, 217–220
Internet, 333–334
intranets, 333–334
networks, 80–81
operating systems, 77–80
passwords, 30
people, 75–77
personalization and configuration information, 217
user IDs, 30
white pages and yellow pages lookups, 75–77
User session data, 340
UTCTime syntax, 183
UTF-8 (universal transfer format -8), 476

V
Values, independently changing, 119
Variables improperly accessed, 119
VBScript, 409
Vendor information, 444–446
Visual Basic, 403
ADSI, 3985
VbX components, 402
Visual C++ with MFC, 403
Visual Java, 403
VPN (virtual private network), 476

W
W3C (World Wide Web Consortium), 403–404
Web applications
n-tier, 90
three-tier, 89
two-tier, 89
Web browsers, 109
Web client APIs, 400–404
Web client-server interaction, 398
Web farms, 165
adding servers, 343–344
Apache Web server, 344
applications, 340
ASP (Active Server Pages), 347
availability, 353–354
backup and restoration of backing store without service interruption, 350
centralized control of constituent servers, 348
CGI applications, 347
crisis notifications, 350
critique, 353–355
design, 340–343
different Web servers handling request, 341
directory configuration, 352
directory infrastructure, 352–352
dispatching browser requests between Web servers, 345
dDNS, 346
hot-plugging and removing servers, 350
HTTP cookies, 342
implementing, 344–347
Intel-based Linux, 344
LDAP directory server, 344–345
limitations, 354–355
load-balancing technique, 341
maintaining state in client browser, 342
maintaining state in DBMS server, 341
management objectives, 347–351
managing, 344
as many IP addresses, 340
mapping of DNS to correct IP addresses, 349
middle-tier applications, 346–347
migrating individual boxes, 351
as one URL, 340
Web farms, continued
  performance monitoring, 349
periodic backup, 347
problem, 339–340
reliability, 353
scalability, 354
schema, 351–352
security, 354
serviceability, 354
stateless applications, 341
three-tier applications, 346
transparent replication of content updates, 349
Web servers, 71
Web protocols, 399–400
Web server APIs
  CGI (common gateway interface), 404–405
  FGI (Fast CGI), 406
  ISAPI, 406
  Java servlets, 406
  NSAPI, 406
  session state, 406–408
Web servers, 71, 397
  ASP (Active Server Pages), 409–410
  authentication, 411
  browser cookies, 407–408
  EJB (Enterprise Java Beans), 409
  hidden text fields, 406
  hybrid session data, 408
  security, 411
  server-side components, 408–411
  SSI (server-side include), 409
  transactions, 408
Web site customer access and update, 311
WebSphere Enterprise Edition, 411
welcom.htm file, 463
White pages, 476
data retrieval, 164–166
lookups, 75–77
Windows 2000
  Active Directory, 67–68
  identity, 154
  Worker threads, 91, 136
Workgroup applications scalability, 109
Workload balancing, 93
Work threads, 118
World Wide Web
  authentication, 411
  projecting static brochures to clients, 397–398
  WSACONNREFUSED message, 230
  WSH (Web Self-Help), 312
    directory infrastructure, 313–315
    enhancements to, 318–319
    operation of, 315–317
    schema, 312–313
X
  X.500, 194, 476
  X.500 data model
    attribute type, 180
    DIT (directory information tree), 180
    schema, 180–181
  X.501 data model, 372
  X.500 directory model, 372
  X.500 family of standards, 32–33, 44
  XML (extensible markup language), 152–153,
  215, 444, 476
  X Series of standards, 32
  XSL (extensible style-sheet language), 152
  X.208 standard, 183
  X.400 standard, 43
  X.500 standard, 42–43, 259
  X.509 standard, 43
  X.520 standard, 43, 211
  X.tpp directory model, 372
Y
  Yellow pages, 476
    data retrieval, 166–168
    lookups, 75–77
  Yhbrid session data, 408
Z
  zJsp, 411