COMPUTER NETWORKS

A Systems Approach

Sponsoring Editor Jennifer Mann
Production Manager Yonie Overton
Production Editor Julie Pabst
Editorial Assistant Jane Elliott
Cover and Text Design Ross Carron Design
Illustration ST Associates, Inc.
Copyeditor Jeff Van Bueren
Proofreader Ken DellaPenta
Composition Ed Sznyter, Babel Press
Indexer Steve Rath
Printer Courier Corporation

Morgan Kaufmann Publishers, Inc.
Editorial and Sales Office
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205 USA
Telephone 415/392-2665
Facsimile 415/982-2665
Internet mkp@mkp.com
Order toll free 800/745-7323

Kurt F. Wendt Library
University of Wisconsin - Madison
215 N. Randall Avenue
Madison, WI 53703-1688

© 1996 by Morgan Kaufmann Publishers, Inc.
All rights reserved
Printed in the United States of America

00 99 98 97 96 5 4 3 2 1

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 written permission of the publisher.

Library of Congress Cataloging-in-Publication Data
Peterson, Larry L.
Computer networks : a systems approach / Larry L. Peterson and Bruce S. Davie.
p. cm.
Includes bibliographical references and index.
ISBN 1-55860-368-9
1. Computer networks. I. Davie, Bruce S. II. Title.
TK5105.5.P479 1996
004.6'5—dc20 96-1590

This material may be protected by Copyright law (Title 17 U.S. Code)

FOREWORD

David Clark
Massachusetts Institute of Technology

The term *spaghetti code* is universally understood as an insult. All good computer scientists worship the god of modularity, since modularity brings many benefits, including the all-powerful benefit of not having to understand all parts of a problem at the same time in order to solve it. Modularity thus plays a role in presenting ideas in a book, as well as in writing code. If a book's material is organized effectively—modularly—the reader can start at the beginning and actually make it to the end.

The field of network protocols is perhaps unique in that the "proper" modularity has been handed down to us in the form of an international standard: the seven-layer reference model of network protocols from the ISO. This model, which reflects a layered approach to modularity, is almost universally used as a starting point for discussions of protocol organization, whether the design in question conforms to the model or deviates from it.

It seems obvious to organize a networking book around this layered model. However, there is a peril to doing so, because the OSI model is not really successful at organizing the core concepts of networking. Such basic requirements as reliability, flow control, or security can be addressed at most, if not all, of the OSI layers. This fact has led to great confusion in trying to understand the reference model. At times it even requires a suspension of disbelief. Indeed, a book organized strictly according to a layered model has some of the attributes of spaghetti code.

Which brings us to this book. Peterson and Davie follow the traditional layered model, but they do not pretend that this model actually helps in the understanding of the big issues in networking. Instead, the authors organize discussion of fundamental concepts in a way that is independent of layering. Thus, after reading the book, readers will understand flow control, congestion control, reliability enhancement, data representation, and synchronization, and will separately understand the implications of addressing these issues in one or another of the traditional layers.

This is a timely book. It looks at the important protocols in use today—especially the Internet protocols. Peterson and Davie have a long involvement in and much experience with the Internet. Thus their book reflects not just the theoretical issues in protocol design, but the real factors that matter in practice. The book looks at some of the protocols that are just emerging now, so the reader can be assured of an up-to-date perspective. But most importantly, the discussion of basic issues is presented in a way that derives from the fundamental

nature of the problem, not the constraints of the layered reference model or the details of today's protocols. In this regard, what this book presents is both timely and timeless. The combination of real-world relevance, current examples, and careful explanation of fundamentals makes this book unique.

CONTENTS

Foreword vii
Preface xvii

1 Foundation
Problem : Building a Network 3
1.1 Motivation 4
1.1.1 Explosive Growth 4
1.1.2 Network Applications 5
1.2 Requirements 10
1.2.1 Connectivity 10
1.2.2 Cost-Effective Resource Sharing 14
1.2.3 Functionality 17
1.2.4 Performance 22
1.3 Network Architecture 29
1.3.1 Layering and Protocols 29
1.3.2 OSI Architecture 35
1.3.3 Internet Architecture 37
1.4 Plan for the Book 39
1.4.1 Chapter Organization 40
1.4.2 Systems Approach 41
1.5 Summary 43
Open Issue : Ubiquitous Networking 43
Further Reading 44
Exercises 46

2 Protocol Implementation
Problem : Protocols Have to Be Implemented 51
2.1 Object-Based Protocol Implementation 52
2.2 Protocols and Sessions 53
2.2.1 Configuring a Protocol Graph 55

2.2.2 Operations on Protocol Objects 56
2.2.3 Operations on Session Objects 57
2.2.4 Process Models for Protocols 59
2.3 Messages 61
2.3.1 Manipulating Messages 62
2.3.2 Implementation 65
2.4 Events 66
2.4.1 Scheduling Timeouts 67
2.4.2 Implementation 67
2.5 Id Map 69
2.5.1 Demultiplexing Packets 70
2.5.2 Implementation 71
2.6 Example Protocol 71
2.6.1 Protocol Semantics 72
2.6.2 Header Files 72
2.6.3 Initialization 74
2.6.4 Opening Connections 75
2.6.5 Demultiplexing Incoming Messages 78
2.6.6 Sending and Receiving Messages 80
2.6.7 Closing the Connection 81
2.7 Summary 82
Open Issue : Cost of Network Software 82
Further Reading 83
Exercises 84

3 Direct Link Networks
Problem : Physically Connecting Hosts 89
3.1 Hardware Building Blocks 90
3.1.1 Nodes 90
3.1.2 Links 91
3.2 Encoding (NRZ, NRZI, Manchester, 4B/5B) 94
3.3 Framing 98
3.3.1 Byte-Oriented Protocols (BISYNC, IMP-IMP, DDCMP) 98
3.3.2 Bit-Oriented Protocols (HDLC) 100
3.3.3 Clock-Based Framing (SONET) 102
3.4 Error Detection 105
3.4.1 Cyclic Redundancy Check 106
3.4.2 Two-Dimensional Parity 108

3.4.3 Internet Checksum Algorithm 109
3.5 Reliable Transmission 110
3.5.1 Stop-and-Wait 111
3.5.2 Sliding Window 113
3.5.3 Concurrent Logical Channels 121
3.6 CSMA/CD (Ethernet) 121
3.6.1 Physical Properties 122
3.6.2 Access Protocol 123
3.6.3 Experience with Ethernet 127
3.7 Token Rings (FDDI) 127
3.7.1 Physical Properties 128
3.7.2 Timed-Token Algorithm 130
3.7.3 Token Maintenance 132
3.7.4 Frame Format 133
3.8 Network Adaptors 134
3.8.1 Components 135
3.8.2 View from the Host 136
3.8.3 Example Device Driver 140
3.9 Summary 143
Open Issue : Does It Belong in Hardware? 144
Further Reading 145
Exercises 146

4 Packet Switching
Problem : Not All Machines Are Directly Connected 151
4.1 Switching and Forwarding 152
4.1.1 Source Routing 154
4.1.2 Virtual Circuit Switching 156
4.1.3 Datagrams 158
4.1.4 Implementation and Performance 161
4.2 Routing 162
4.2.1 Network as a Graph 163
4.2.2 Distance Vector 164
4.2.3 Link State 169
4.2.4 Metrics 174
4.2.5 Routing, Addressing, and Hierarchy 176
4.3 Cell Switching (ATM) 177
4.3.1 Cells 177

4.3.2 Segmentation and Reassembly 182
4.3.3 Virtual Paths 186
4.3.4 Physical Layers for ATM 187
4.4 Switching Hardware 188
4.4.1 Design Goals 188
4.4.2 Ports and Fabrics 190
4.4.3 Crossbar Switches 192
4.4.4 Self-Routing Fabrics 196
4.4.5 Shared-Media Switches 200
4.5 Summary 202
Open Issue : What Makes a Route Good? 202
Further Reading 203
Exercises 204

5 Internetworking 209
Problem : There Is More than One Network 210
5.1 Bridges and Extended LANs 210
5.1.1 Learning Bridges 211
5.1.2 Spanning Tree Algorithm 215
5.1.3 Broadcast and Multicast 216
5.1.4 Limitations of Bridges 217
5.2 Simple Internetworking (IP) 217
5.2.1 What Is an Internetwork? 219
5.2.2 Service Model 229
5.2.3 Global Addresses 231
5.2.4 Datagram Forwarding in IP 233
5.2.5 Address Translation (ARP) 236
5.2.6 Error Reporting (ICMP) 237
5.3 Global Internet 239
5.3.1 Subnetting 243
5.3.2 Route Propagation (RIP, OSPF, BGP) 250
5.3.3 Classless Routing (CIDR) 252
5.4 Next Generation IP (IPv6) 253
5.4.1 Historical Perspective 254
5.4.2 Addresses and Routing 258
5.4.3 IPv6 Features 262
5.5 Multicast 263
5.5.1 Link-State Multicast

Ex.1022 / Page 8 of 63
Intel Corporation and Lenovo (United States) Inc.

5.5.2 Distance-Vector Multicast 264
5.6 Host Names (DNS) 267
5.6.1 Domain Hierarchy 268
5.6.2 Name Servers 269
5.6.3 Name Resolution 272
5.7 Summary 274
Open Issue : IP versus ATM 276
Further Reading 277
Exercises 278

6 End-to-End Protocols
Problem : Getting Processes to Communicate 283
6.1 Simple Demultiplexer (UDP) 284
6.2 Reliable Byte Stream (TCP) 285
6.2.1 End-to-End Issues 286
6.2.2 Segment Format 289
6.2.3 Connection Establishment and Termination 292
6.2.4 Sliding Window Revisited 296
6.2.5 Adaptive Retransmission 301
6.2.6 Record Boundaries 304
6.3 Remote Procedure Call 305
6.3.1 Bulk Transfer (BLAST) 309
6.3.2 Request/Reply (CHAN) 315
6.3.3 Dispatcher (SELECT) 324
6.3.4 Putting It All Together (SunRPC) 326
6.4 Application Programming Interface 332
6.5 Performance 335
6.5.1 Experimental Method 335
6.5.2 Latency 336
6.5.3 Throughput 337
6.6 Summary 338
Open Issue : Application-Specific Protocols 339
Further Reading 340
Exercises 341

7 End-to-End Data
Problem : What Do We Do with the Data? 347
7.1 Presentation Formatting 348

7.1.1 Taxonomy 349
7.1.2 Examples (XDR, ASN.1, NDR) 353
7.2 Data Compression 357
7.2.1 Lossless Compression Algorithms 358
7.2.2 Image Compression (JPEG) 360
7.2.3 Video Compression (MPEG) 364
7.3 Security 368
7.3.1 Secret Key versus Public Key Cryptography 369
7.3.2 Encryption Algorithms (DES, RSA) 371
7.3.3 Authentication Protocols (Kerberos) 378
7.3.4 Message Integrity Protocols 382
7.4 Summary 385
Open Issue : Presentation--Layer of the 90s 386
Further Reading 387
Exercises 388

8 Congestion Control
Problem : Allocating Resources 393
8.1 Issues in Congestion Control 394
8.1.1 Network Model 394
8.1.2 Taxonomy 397
8.1.3 Evaluation Criteria 399
8.2 Queuing Disciplines 401
8.2.1 FIFO 401
8.2.2 Fair Queuing 402
8.3 TCP Congestion Control 406
8.3.1 Additive Increase/Multiplicative Decrease 406
8.3.2 Slow Start 409
8.3.3 Fast Retransmit and Fast Recovery 413
8.4 Congestion-Avoidance Mechanisms 415
8.4.1 DECbit 415
8.4.2 RED Gateways 417
8.4.3 Source-Based Congestion Avoidance 419
8.5 Virtual Clock 424
8.5.1 Defining Rate 425
8.5.2 Virtual Clock as a Queuing Discipline 426
8.5.3 Virtual Clock as a Flow Meter 428
8.5.4 Other Rate-Based Mechanisms (ABR) 428

8.6 Summary 430
Open Issue : Inside versus Outside the Network 430
Further Reading 431
Exercises 432

9 High-Speed Networking
Problem : What Breaks When We Go Faster? 437
9.1 Latency Issues 438
9.1.1 Latency/Throughput Tradeoff 438
9.1.2 Implications 439
9.2 Throughput Issues 441
9.2.1 Memory Bottleneck 442
9.2.2 Techniques for Avoiding Data Transfers 443
9.3 Integrated Services 450
9.3.1 New Service Model 451
9.3.2 Mechanisms (RSVP) 456
9.4 Summary 464
Open Issue : Realizing the Future 465
Further Reading 465
Exercises 466

APPENDIX Network Management
A.1 SNMP Overview 472
A.2 MIB Variables 473
A.2.1 System Group 474
A.2.2 Interfaces Group 476
A.2.3 Address Translation Group 477
A.2.4 IP Group 478
A.2.5 TCP Group 484
A.2.6 UDP Group 488

Glossary 491
References 511
Index 521

PREFACE

Without question, computer networking is one of the most exciting computer science topics to emerge in recent years. The media have done a more than adequate job of popularizing networking, and examples of its rapid spread are plentiful: *USA Today* regularly publishes the addresses of interesting World Wide Web sites; we are encouraged to send comments to *NBC Nightly News* via Internet mail; both the White House and Congress are online (as is Her Majesty's Treasury); and, perhaps most impressively, it is possible to order pizza over the Internet. Surely any technology that receives this much attention must be worth understanding.

The concern, of course, is that there might be little or no intellectual substance once we get beyond the buzzwords. Fortunately, this is not the case. When we peel away all the hype, the field of computer networking contains a wealth of interesting problems and clever solutions founded on solid underlying principles. The challenge is to identify and understand these principles. That is why we decided to write this book. Our intent is that the book should serve as the text for an introductory networking class, at either the graduate or upper-division undergraduate level. We also believe that the book's focus on core concepts should be appealing to industry professionals who are retraining for network-related assignments, as well as current network practitioners who want to understand the "whys" behind the protocols they work with every day.

Approach

It is our experience that people learning about networks for the first time often have the impression that network protocols are some sort of edict handed down from on high, and that their job is to learn as many TLAs (Three Letter Acronyms) as possible. In fact, protocols are the building blocks of a complex system developed through the application of engineering design principles. Moreover, they are constantly being refined, extended, and replaced based on real-world experience. With this in mind, our goal with this book is to do more than survey the protocols in use today. Instead, we explain the underlying principles of sound network design.

This goal implies several things about our approach. Perhaps most importantly, the book takes a systems-oriented view of computer networking. What we mean by this is that the book identifies the underlying building blocks, shows how they can be glued together to con-

struct a complete network, measures the resulting system experimentally, and uses this data to quantitatively analyze various design options. This is in contrast to looking at networking through the eyes of a standardization body, a telephone company executive, or a queuing theorist.

In keeping with our systems perspective, this book explains *why* networks are designed the way they are. It does this by starting from first principles, rather than by simply marching through the set of protocols that happen to be in use today. Moreover, we do not adhere to a rigidly layerist point of view, but instead allow ourselves to address issues that cannot be neatly packaged in any single layer. It is our experience that once you understand the underlying concepts, any new protocol that you are confronted with—especially one that cannot be pigeonholed in an existing layer—will be relatively easy to digest. Given the volatility and explosive growth of networking in the 1990s, this focus on principles rather than artifacts is well justified.

The book does two things to help make these underlying concepts more tangible. First, it draws heavily from the lessons of the Internet. As a functioning, global network, the Internet is rich with examples that illustrate how networks are designed in practice. This book exploits the Internet experience in several ways: as a road map that guides the organization of this book, as a source of examples that illustrate the fundamental ideas underlying computer networks, and as a set of design principles that color our way of looking at computer networks. Second, the book includes code segments from a working network subsystem—the *x*-kernel. The book is not an exhaustive inspection of the code. Rather, snippets of *x*-kernel code are given throughout the book to help make the discussion more concrete. The basic strategy is to provide you with not only an understanding of the components of a network, but also a feel for how all the pieces are put together.

How to Use This Book

The chapters of this book can be organized into three groups:

- Introductory material: Chapters 1–2.
- Core topics: Chapters 3–6.
- Advanced topics: Chapters 7–9.

For an undergraduate course, extra class time will most likely be needed to help students digest the introductory material in the first two chapters, probably at the expense of the more advanced topics covered in Chapters 7 through 9. In contrast, the instructor for a graduate course should be able to cover the first two chapters in only a few lectures—with students studying the material more carefully on their own—thereby freeing up additional class time to cover the last three chapters in depth. Both graduate and undergraduate classes will want to cover the core material contained in the middle four chapters, although an undergraduate class might choose to skim the more advanced sections (e.g., 3.2, 3.8, 4.4, and 5.5).

Ex.1022 / Page 13 of 63
Intel Corporation and Lenovo (United States) Inc.

For those of you using the book in self-study, we believe that the topics we have selected cover the core of computer networking, and so we recommend that the book be read sequentially, from front to back. In addition, we have included a liberal supply of references to help you locate supplementary material that is relevant to your specific areas of interest.

There are several other things to note about the topic selection and organization. First, the book takes a unique approach to the topic of congestion control by pulling all topics related to congestion control and resource allocation together in a single place—Chapter 8. We do this because the problem of congestion control cannot be solved at any one level, and we want you to consider the various design options at the same time. (This is consistent with our view that strict layering often obscures important design tradeoffs.) A more traditional treatment of congestion control is possible, however, by studying Section 8.2 in the context of Chapter 4 and Section 8.3 in the context of Chapter 6.

Second, even though Chapter 9 is the only chapter explicitly labeled *High-Speed Networking*, the topic is covered in substantial depth throughout the book. For example, Section 3.8 discusses the role played by network adaptors in delivering high throughput, 4.3 describes ATM networks, 4.4 describes hardware switches used in high-speed networks, 7.2 discusses MPEG video compression, and 8.5 discusses resource reservation. In addition, video applications are used throughout the book to motivate various design decisions. In short, high-speed networking is becoming commonplace, and we believe that it no longer deserves to be treated purely as an advanced topic.

Third, example code from the *x*-kernel is used throughout the book to illustrate important concepts, with an overview of the *x*-kernel given in Chapter 2. How much time you spend studying Chapter 2 depends, of course, on how you plan to use the *x*-kernel throughout the rest of the book. We envision the code being used at one of three levels.

- The code can be *ignored*. Each of the chapters contains a complete discussion of the topic at hand; the code simply punctuates this discussion with tangible evidence that the ideas being presented can be put to practice.
- The code can be *read*. While this certainly involves some effort, the investment is relatively low (the *x*-kernel was explicitly designed to support network software and so has a high "signal-to-noise" ratio) and the payoff is real (you can see firsthand how the underlying ideas are realized in practice).
- The code can be *written*. Because the complete code is available online, it can be used to provide hands-on experience. Some of the exercises at the end of each chapter suggest programming assignments that you can do using the *x*-kernel.

Additional information on how (and why) to use the *x*-kernel is given on page xx.

Finally, the book has several features that we encourage you to take advantage of.

- Each chapter begins with a *Problem* that identifies the next set of issues that must be addressed in the design of a network. Each problem statement introduces the issues

Ex.1022 / Page 14 of 63
Intel Corporation and Lenovo (United States) Inc.

to be explored in the chapter that follows and outlines individual topics covered in that chapter.

- *Shaded sidebars* throughout the text elaborate on the topic being discussed or introduce a related advanced topic. In many cases, these sidebars relate real-world anecdotes about networking.
- Key paragraphs are *highlighted* and marked in the margin with an icon. These paragraphs often summarize an important nugget of information that we want you to take away from the discussion. In many cases, this nugget is a system design principle that you will see applied in many different situations.
- Even though the book's focus is on core concepts rather than existing protocol specifications, real protocols are used to illustrate most of the important ideas. As a result, the book can be used as a source of reference for these example protocols. To help you find the descriptions of the protocols, each applicable section heading parenthetically identifies the protocols defined in that section. For example, Section 6.2, which describes the principles of reliable end-to-end protocols, provides a detailed description of TCP, the canonical example of such a protocol.
- We conclude the main body of each chapter with an *Open Issue.* The idea of this section is to expose you to various unresolved issues that are currently being debated in the research community, the commercial world, and/or society as a whole. We have found that discussing these issues helps to make the subject of networking more relevant and exciting.
- As part of the Further Reading section at the end of each chapter, a specific *recommended reading list* is given. This list generally contains the seminal papers on the topics discussed in the chapter. We strongly recommend that advanced readers (e.g., graduate students) study the papers in this reading list to supplement the material covered in the chapter.

Software

As mentioned above, this book uses the x-kernel to illustrate how network software is implemented. The x-kernel was designed to support the rapid implementation of efficient network protocols. A protocol implementation can be done rapidly because the x-kernel provides a set of high-level abstractions that are tailored specifically to support protocol implementations. x-kernel protocols are efficient because these abstractions are themselves implemented using highly optimized data structures and algorithms. In fact, the world speed record for delivering network data from an ATM network into a desktop workstation—516 Mbps—is held by the x-kernel at the time of this printing.

Because the x-kernel has the dual goals of running fast and making it easy to implement protocols, it has also proved to be an invaluable pedagogical tool, one that is used extensively in

the *Principles of Computer Networking* class at the University of Arizona. There are two aspects to the pedagogical value of the x-kernel. First, because the x-kernel defines a concise framework for implementing protocols, it can be viewed as simply codifying the essential elements of network protocols. An intimate understanding of the interfaces provided by the x-kernel necessarily means appreciating the mechanisms found in all protocols. Second, it provides a convenient means for students to get hands-on experience with network protocols—students are able both to implement protocols and to evaluate their performance.

On the subject of "hands-on experience," many of the exercises at the end of each chapter suggest programming assignments that can be done, some (but not all) of which use the x-kernel. Typically, these assignments take on one of the following four forms:

- Write a small program that tests or measures some narrow aspect of a network system. These assignments are independent of the x-kernel.
- Run and measure existing protocols currently distributed with the x-kernel. Sometimes these assignments involve making simple modifications to existing x-kernel code.
- Implement and evaluate standard protocols based on existing protocol specifications. These assignments use the x-kernel as a protocol implementation framework.
- Design, implement, and evaluate new protocols from scratch. These assignments use the x-kernel as a protocol implementation framework.

For these assignments, students typically run the x-kernel as a user process on Unix workstations connected by an Ethernet. Several flavors of Unix are currently supported, including SunOS, Solaris, OSF/1, IRIX, and Linux. In this case, the x-kernel generates packets that are delivered between machines over the Ethernet. An alternative is to run the x-kernel in *simulation mode*, again on top of Unix. In this case, networks with arbitrary topology, bandwidth, and delay characteristics can be simulated as a single Unix process.

The x-kernel source code is available via anonymous `ftp` to `cs.arizona.edu`, `cd`'ing to directory `xkernel`, and accessing file `xkernel.tar.Z`. You can also find the x-kernel home page on the Web at URL

`http://www.cs.arizona.edu/xkernel`

This page includes a link to online documentation, as well as instructions on how to obtain help with the x-kernel. Additional information about the x-kernel and how it can be used in conjunction with this book can be found at URL

`http://www.mkp.com`

Acknowledgments

This book would not have been possible without the help of so many people. We would like to thank them for their efforts in improving the end result. Before we do so, however, we

should mention that we have done our best to correct the mistakes that the reviewers have pointed out and to accurately describe the protocols and mechanisms that our collegues have explained to us. We alone are responsible for any remaining errors. If you should find any of these, please send email to our publisher, Morgan Kaufmann, at netbugs@mkp.com, and we will endeavor to correct them in future printings of this book.

First and foremost, we would like to thank our series editor, Dave Clark. His involvement in the series played no small part in our decision to write this book with Morgan Kaufmann, and the suggestions, expertise, perspective, and encouragement he offered us throughout the writing process proved invaluable.

Several members of the Network Systems Research Group at the University of Arizona contributed ideas, examples, corrections, data, and code to this book. Andy Bavier fleshed out many of the exercises and examples, and he offered helpful suggestions on the clarity of the early drafts. Lawrence Brakmo answered countless questions about TCP and provided the data used in the figures on TCP congestion control. David Mosberger served as the resident expert on MPEG and JPEG, and he ran many of the performance experiments reported in the book. Andy, Lawrence, and David also wrote most of the x-kernel code fragments that ended up in the book. Richard Schroeppel, Hilarie Orman, and Sean O'Malley were valuable resources on network security and supplied the security-related performance numbers. Sean also helped us organize our thoughts on presentation formatting. Finally, countless people have contributed to the x-kernel over the years. We would especially like to thank (in addition to those mentioned above) Ed Menze, Sandy Miller, Hasnain Karampurwala, Mats Bjorkman, Peter Druschel, Erich Nahum, and Norm Hutchinson.

Several other people were kind enough to answer our questions and correct our misunderstandings. Rich Salz and Franco Travostino from OSF taught us about NDR. Jeffrey Mogul from Digital's Western Research Lab and John Wroclawski from MIT helped us understand how fast workstations can switch packets. Gregg Townsend from the University of Arizona explained how GIF works. Gail Lalk and Tom Robe of Bellcore answered questions about SONET.

We would also like to thank the many people who reviewed drafts of various chapters. The list is long, and includes Chris Edmondson-Yurkanan of the University of Texas at Austin, Willis Marti of Texas A&M University, Frank Young of Rose-Hullman Institute of Technology, Tom Robertazzi of SUNY Stony Brook, Jon Crowcroft of University College London, Eric Crawley of Bay Networks, Jim Metzler of Strategic Networks Consulting, Dan McDonald of the Navy Research Lab, Mark Abbott of IBM, Tony Bogovic of Bellcore, David Hayes of San Jose State University, S. Ron Oliver of California State Polytechnic University-Pomona, Donald Carpenter of the University of Nebraska at Kearney, and Ski Illnicki of Hewlett-Packard. Their comments and suggestions were invaluable. Also, a special thanks to Professors Peter Druschel of Rice University, Izidor Gertner of CCNY, and Mansoor Alam of the University of Toledo for their feedback based on class-testing the first draft of the book.

We also want to thank the Advanced Research Projects Agency and the National Science Foundation for supporting our research over the past several years. That research helped to shape our perspective on networking, as well as to produce tangible results like the x-kernel. Thanks also to Bell Communications Research, which supported some of our research and writing.

We would like to thank all the people at Morgan Kaufmann who helped shepherd us through the book-writing process, especially our sponsoring editor, Jennifer Mann, our production editor, Julie Pabst, and our production manager, Yonie Overton. We cannot imagine a more enthusiastic or harder-working group of people.

Finally, we wish to thank our long-suffering wives, Lynn Peterson and Jody Davie, who put up with our complete obsession with the book (and consequent neglect of our families), especially in the final months of writing.

Ex.1022 / Page 18 of 63
Intel Corporation and Lenovo (United States) Inc.

PROBLEM

What Do We Do with the Data?

It is a capital mistake to theorize before one has data.

—Conan Doyle

Our focus up to this point has been on the hierarchy of protocols that deliver messages from one process to another. What we sometimes forget is that these messages carry data—information that is somehow meaningful to the application processes using the network. While this data is by nature application-specific, there are certain common transformations that application programs apply to their data before they send it. This chapter considers three such transformations: presentation formatting, compression, and encryption. These transformations are sometimes called *data manipulations.*

Presentation formatting has to do with translating data from the representation used internally by the application program into a representation that is suitable for transmission over a network. It is concerned with how different data types (e.g., integers, floating-point numbers, character strings) are encoded in messages and with the fact that different machines typically use different encodings. Presentation formatting also deals with the representation of complex data structures (which may include pointers that have only local significance) in a way that they can be transmitted to a remote machine.

Compression is concerned with reducing the number of bits it takes to represent a particular piece of data, with the goal being to transmit the data using as little network bandwidth as possible. Data is typically compressed before it is given to the transport protocol on the sending side and then decompressed on the receiving side before the application begin to process it.

The third transformation, *encryption*, is used to ensure that anyone who might be eavesdropping on the network—that is, snooping packets as they fly by on a shared-access network or as they sit in a router queue—is not able to read the data in the message. Encryption is also used to authenticate participants and to ensure message integrity.

One of the important aspects of the data manipulations studied in this chapter is that they involve processing every byte of data in the message. This is in contrast to most of the protocols we have seen up to this point, which process a mes-

sage without ever looking at its contents. Because of this need to read, compute on, and write every byte of data in a message, data manipulations strongly affect the end-to-end throughput we can achieve over a network. In fact, these manipulations are often the limiting factor.

Note that there is a fourth common function that processes every byte of data in a message—error checking. We considered error checking in an earlier chapter (Section 3.4) because it is usually implemented at lower levels of the protocol hierarchy; it is typically not implemented above the end-to-end protocol.

7

End-to-End Data

7.1 Presentation Formatting

From the network's perspective, application programs send messages to each other. From the application's perspective, however, these messages contain various kinds of *data*—arrays of integers, video frames, lines of text, digital images, and so on. For some of this data, well-established formats have been defined. For example, video is typically transmitted in Moving Pictures Experts Group (MPEG) format, still images are usually transmitted in Joint Photographic Experts Group (JPEG) or Graphical Interchange Format (GIF) format, multimedia documents are transmitted in HyperText Markup Language (HTML) or Multipurpose Internet Mail Extensions (MIME) format, and text is typically transmitted in American Standard Code for Information Interchange (ASCII) format.[1] For other types of data, however, there is no universal agreement about the format; each computer defines its own representation. This is certainly true for the kinds of data that regular programs compute on, such as integers, floating-point numbers, character strings, arrays, and structures.

One of the most common transformations of network data is from the representation used by the application program into a form that is suitable for transmission over a network and vice versa. This transformation is typically called *presentation formatting*. As illustrated in Figure 7.1, the sending program translates the data it wants to transmit from the representation it uses internally into a message that can be transmitted over the network; that is, the data is *encoded* in a message. On the receiving side, the application translates this arriving message into a representation that it can then process; that is, the message is *decoded*. Encoding is sometimes called *argument marshalling*, and decoding is sometimes called *unmarshalling*. This terminology comes from the RPC world, where the client thinks it is invoking a procedure with a set of arguments, but these arguments are then "brought together and ordered in an appropriate and effective way"[2] to form a network message.

You might ask what makes this problem challenging enough to warrant a name like marshalling. One reason is that computers represent data in different ways. For example, some computers represent floating-point numbers in IEEE standard 754 format, while other machines still use their own nonstandard format. Even for something as simple as integers, different architectures use different sizes (e.g., 16 bit, 32 bit, 64 bit). To make matters worse, on some machines integers are represented in *big-endian* form (the most significant bit of a word is in the byte with the highest address), while on other machines integers are represented in *little-endian* form (the most significant bit is in the byte with the lowest address). The Motorola 680x0 is an example of a big-endian architecture and the Intel 80x86 is an example of a little-endian architecture. For example, the big-endian and little-endian representation of the integer 34,677,374 is given in Figure 7.2.

Another reason that marshalling is difficult is that application programs are written in different languages, and even when you are using a single language, there may be more than

[1] We will study GIF, JPEG, and MPEG in Section 7.2, in the context of data compression. In effect, however, these standards combine elements of presentation formatting and data compression.
[2] This is a definition of *marshalling* taken from Webster's New Collegiate Dictionary.

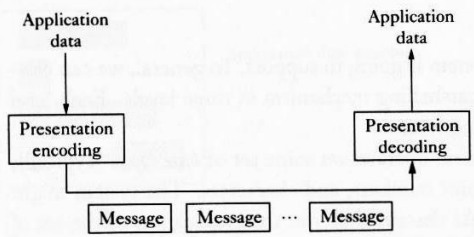


Figure 7.1 Presentation formatting involves encoding and decoding application data.

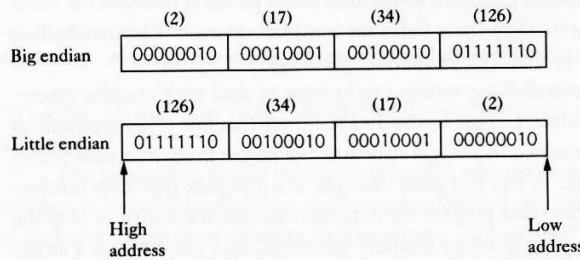


Figure 7.2 Big-endian and little-endian byte order for the integer 34,677,374.

one compiler. For example, compilers have a fair amount of latitude in how they lay out structures (records) in memory, such as how much padding they put between the fields that make up the structure. Thus, you could not simply transmit a structure from one machine to another, even if both machines were of the same architecture and the program was written in the same language, because the compiler on the destination machine might align the fields in the structure differently.

7.1.1 Taxonomy

While anyone who has worked on argument marshalling would tell you that no rocket science is involved—it is a small matter of bit twiddling—there are a surprising number of design choices that you must address. We begin by giving a simple taxonomy for argument marshalling systems. The following is by no means the only viable taxonomy, but it is sufficient to cover most of the interesting alternatives.

Data Types

The first question is what data types the system is going to support. In general, we can classify the types supported by an argument marshalling mechanism at three levels. Each level complicates the task faced by the marshalling system.

At the lowest level, a marshalling system operates on some set of *base types*. Typically, the base types include integers, floating-point numbers, and characters. The system might also support ordinal types and booleans. As described above, the implication of the set of base types is that the encoding process must be able to convert each base type from one representation to another, for example, convert an integer from big-endian to little-endian.

At the next level are *flat types*—structures and arrays. While flat types might at first not appear to complicate argument marshalling, the reality is that they do. The problem is that the compilers used to compile application programs sometimes insert padding between the fields that make up the structure so as to align these fields on word boundaries. The marshalling system typically *packs* structures so that they contain no padding.

At the highest level, the marshalling system might have to deal with *complex types*—those types that are built using pointers. That is, the data structure that one program wants to send to another might not be contained in a single structure, but might instead involve pointers from one structure to another. A tree is a good example of a complex type that involves pointers. Clearly, the data encoder must prepare the data structure for transmission over the network because pointers are implemented by memory addresses, and just because a structure lives at a certain memory address on one machine does not mean it will live at the same address on another machine. In other words, the marshalling system must *linearize* (flatten) complex data structures.

In summary, depending on how complicated the type system is, the task of argument marshalling usually involves converting the base types, packing the structures, and linearizing the complex data structures, all to form a contiguous message that can be transmitted over the network. Figure 7.3 illustrates this task.

Conversion Strategy

Once the type system is established, the next issue is what conversion strategy the argument marshaller will use. There are two general options: *canonical intermediate form* and *receiver-makes-right*. We consider each, in turn.

The idea of canonical intermediate form is to settle on an external representation for each type; the sending host translates from its internal representation to this external representation before sending data, and the receiver translates from this external representation into its local representation when receiving data. To illustrate the idea, consider integer data; other types are treated in a similar manner. You might declare that the big-endian format will be used as the external representation for integers. The sending host must translate each integer it sends into big-endian form and the receiving host must translate big-endian integers into whatever representation it uses. (This is what is done in the Internet for protocol headers.)

Ex.1022 / Page 23 of 63
Intel Corporation and Lenovo (United States) Inc.

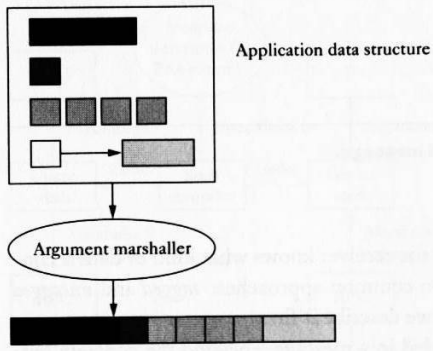


Figure 7.3 Argument marshalling: converting, packing, and linearizing.

Of course, a given host might already use big-endian form, in which case no conversion is necessary.

The alternative, which is sometimes called receiver-makes-right, has the sender transmit data in its own internal format; the sender does not convert the base types, but usually has to pack and flatten more complex data structures. The receiver is then responsible for translating the data from the sender's format into its own local format. The problem with this strategy is that every host must be prepared to convert data from all other machine architectures. In networking, this is known as an *N-by-N solution*: each of N machine architectures must be able to handle all N architectures. In contrast, in a system that uses a canonical intermediate form, each host needs to know only how to convert between its own representation and a single other representation—the external one.

Using a common external format is clearly the correct thing to do, right? This has certainly been the conventional wisdom in the networking community for the past 20 years. The answer is not cut and dried, however. It turns out that there are not that many different representations for the various base classes, or said another way, N is not that large. In addition, the most common case is for two machines of the same type to be communicating with each other. In this situation, it seems silly to translate data from that architecture's representation into some foreign external representation, only to have to translate the data back into the same architecture's representation on the receiver.

A third option, although we know of no existing system that exploits it, is to use receiver-makes-right if the sender knows that the destination has the same architecture; the sender would use some canonical intermediate form if the two machines use different architectures. How would a sender learn the receiver's architecture? It could either learn this information from a name server, or by first using a simple test case to see if the appropriate result occurs.

Ex.1022 / Page 24 of 63
Intel Corporation and Lenovo (United States) Inc.

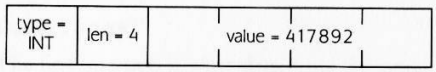


Figure 7.4 A 32-bit integer encoded in a tagged message.

Tags

The third issue in argument marshalling is how the receiver knows what kind of data is contained in the message it receives. There are two common approaches: *tagged* and *untagged* data. The tagged approach is most intuitive, so we describe it first.

A tag is any additional information included in a message—beyond the concrete representation of the base types—that helps the receiver decode the message. There are several possible tags that might be included in a message. For example, each data item might be augmented with a *type* tag. A type tag indicates that the value that follows is an integer, a floating-point number, or whatever. Another example is a *length* tag. Such a tag is used to indicate the number of elements in an array or the size of an integer. A third example is an *architecture* tag, which might be used in conjunction with the receiver-makes-right strategy to specify the architecture on which the data contained in the message was generated. Figure 7.4 depicts how a simple 32-bit integer might be encoded in a tagged message.

The alternative, of course, is not to use tags. How does the receiver know how to decode the data in this case? It knows because it was programmed to know. In other words, if you call a remote procedure that takes two integers and a floating-point number as arguments, then there is no reason for the remote procedure to inspect tags to know what it has just received. It simply assumes that the message contains two integers and a float, and decodes it accordingly. Note that while this works for most cases, the one place it breaks down is when sending variable-length arrays. In such a case, a length tag is commonly used to indicate how long the array is.

It is also worth noting that the untagged approach means that the presentation formatting is truly end to end. It is not possible for some intermediate agent to interpret the message unless the data is tagged. Why would an intermediate agent need to interpret a message, you might ask? Stranger things have happened, mostly resulting from ad hoc solutions to unexpected problems that the system was not engineered to handle. Poor network design is beyond the scope of this book.

Stubs

A stub is the piece of code that implements argument marshalling. Stubs are typically used to support RPC. On the client side, the stub marshals the procedure arguments into a message that can be transmitted by means of the RPC protocol. On the server side, the stub converts the message back into a set of variables that can be used as arguments to call the remote procedure. Stubs can either be interpreted or compiled.

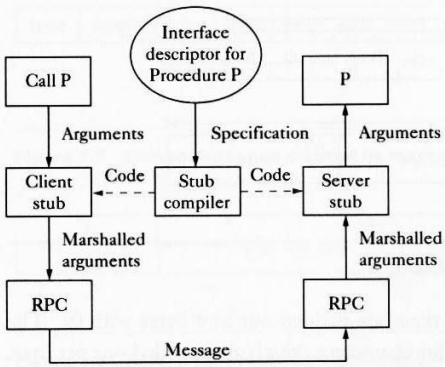


Figure 7.5 Stub compiler takes interface description as input and outputs client and server stubs.

In a compilation-based approach, each procedure has a "customized" client and server stub. This stub is typically generated by a stub compiler, based on a description of the procedure's interface. This situation is illustrated in Figure 7.5. Since the stub is compiled, it is usually very efficient. In an interpretation-based approach, the system provides "generic" client and server stubs that have their parameters set by a description of the procedure's interface. Because it is easy to change this description, interpreted stubs have the advantage of being flexible. Compiled stubs are more common in practice.

7.1.2 Examples (XDR, ASN.1, NDR)

We now briefly describe three popular network data representations in terms of this taxonomy. We use the integer base type to illustrate how each system works.

XDR

External Data Representation (XDR) is the network format used with SunRPC. It supports the C type system but without function pointers, it defines a canonical intermediate form, it does not use tags (except to indicate array lengths), and its stubs are compiled.

An XDR integer is a 32-bit data item that encodes a C integer. It is represented in twos complement notation, with the most significant byte of the C integer in the first byte of the XDR integer, and the least significant byte of the C integer in the fourth byte of the XDR integer. That is, XDR uses big-endian format for integers. XDR supports both signed and unsigned integers, just as C does.

XDR represents variable-length arrays by first specifying an unsigned integer (4 bytes) that gives the number of elements in the array, followed by that many elements of the appropriate type. XDR encodes the components of a structure in the order of their declaration in the structure. For both arrays and structures, the size of each element/component is repre-

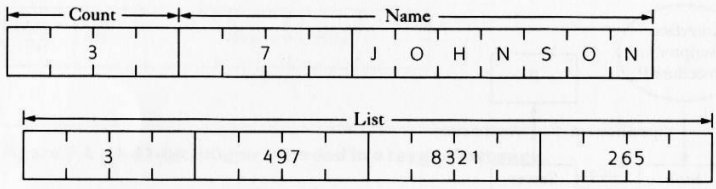


Figure 7.6 Example encoding of a structure in XDR.

sented in a multiple of 4 bytes. Smaller data types are padded out to 4 bytes with 0s. The exception to this "pad to 4 bytes" rule is made for characters, which are encoded one per byte.

The following code fragment gives an example C structure (item) and the XDR routine that encodes/decodes this structure (xdr_item). Figure 7.6 schematically depicts XDR's on-the-wire representation of this structure when the field name is seven characters long and the array list has three values in it.

In this example, xdr_array, xdr_int, and xdr_string are three primitive functions provided by XDR to encode and decode arrays, integers, and character strings, respectively. Argument xdrs is a "context" variable that XDR uses to keep track of where it is in the message being processed; it includes a flag that indicates whether this routine is being used to encode or decode the message. In other words, routines like xdr_item are used on both the client and the server. Note that the application programmer can either write the routine xdr_item by hand or use a stub compiler called rpcgen (not shown) to generate this encoding/decoding routine. In the latter case, rpcgen takes the remote procedure that defines the data structure item as input, and outputs the corresponding stub.

```
#define MAXNAME 256;
#define MAXLIST 100;

struct item {
   int     count;
   char    name[MAXNAME];
   int     list[MAXLIST];
};

bool_t
xdr_item(XDR *xdrs, struct item *ptr)
{
    return(xdr_int(xdrs, &ptr->count) &&
       xdr_string(xdrs, &ptr->name, MAXNAME) &&
       xdr_array(xdrs, &ptr->list, &ptr->count, MAXLIST, sizeof(int), xdr_int));
}
```

Exactly how XDR performs depends, of course, on the complexity of the data. In a simple case of an array of integers, where each integer has to be converted from one byte order to

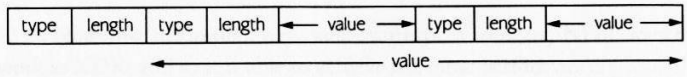


Figure 7.7 Compound types created by means of nesting in ASN.1/BER.

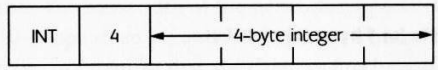


Figure 7.8 ASN.1/BER representation for a 4-byte integer.

another, an average of 2.75 instructions are required for each byte on a DEC Alpha, meaning that converting the whole array is a memory-bound operation. On the 175-MHz Alpha we have been using throughout this book, for example, this means an upper limit on the order of 70 MBps. More complex conversions that require more instructions per byte will run slower.

ASN.1

Abstract Syntax Notation One (ASN.1) is an ISO standard that defines, among other things, a representation for data sent over a network. The representation-specific part of ASN.1 is called the Basic Encoding Rules (BER). ASN.1 supports the C type system without function pointers, defines a canonical intermediate form, uses type tags, and its stubs can be either interpreted or compiled.

ASN.1 represents each data item with a triple of the form

⟨ tag, length, value ⟩.

The tag is typically an 8-bit field, although ASN.1 allows for the definition of multi-byte tags. The length field specifies how many bytes make up the value; we discuss length more below. Compound data types, such as structures, can be constructed by nesting primitive types, as illustrated in Figure 7.7.

If the value is 127 or fewer bytes long, then the length is specified in a single byte. Thus, for example, a 32-bit integer is encoded as a 1-byte type, a 1-byte length, and the 4 bytes that encode the integer, as illustrated in Figure 7.8. The value itself, in the case of an integer, is represented in twos complement notation and big-endian form, just as in XDR. Keep in mind that even though the value of the integer is represented in exactly the same way in both XDR and ASN.1, the XDR representation has neither the type nor the length tags associated with that integer. These two tags both take up space in the message, and more importantly, require processing during marshalling and unmarshalling. This is one reason that ASN.1 is not as efficient as XDR. Another is that the very fact that each data value is preceded by a length field means that the data value is unlikely to fall on a natural byte boundary (e.g., an integer beginning on a word boundary). This complicates the encoding/decoding process.

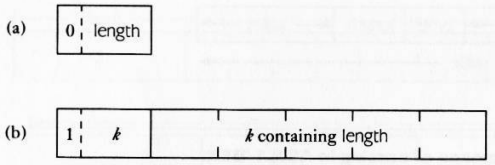


Figure 7.9 ASN.1/BER representation for length: (a) 1 byte; (b) multibyte.

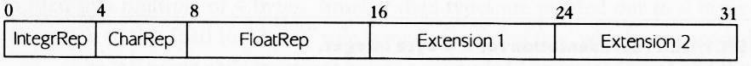


Figure 7.10 NDR's architecture tag.

If the value is 128 or more bytes long, then multiple bytes are used to specify its length. At this point you may be asking why a byte can specify a length of up to 127 bytes rather than 256. The reason is that 1 bit of the length field is used to denote how long the length field is. A 0 in the eighth bit indicates a 1-byte length field. To specify a longer length, the eighth bit is set to 1, and the other 7 bits indicate how many additional bytes make up the length. Figure 7.9 illustrates a simple 1-byte length and a multibyte length.

NDR

Network Data Representation (NDR) is the data-encoding standard used in the Distributed Computing Environment (DCE).[3] Unlike XDR and ASN.1, NDR uses receiver-makes-right. It does this by inserting an architecture tag at the front of each message; individual data items are untagged. NDR uses a compiler to generate stubs. This compiler takes a description of a program written in the Interface Definition Language (IDL) and generates the necessary stubs. IDL looks pretty much like C, and so essentially supports the C type system.

Figure 7.10 illustrates the 4-byte architecture-definition tag that is included at the front of each NDR-encoded message. The first byte contains two 4-bit fields. The first field, IntegrRep, defines the format for all integers contained in the message. A 0 in this field indicates big-endian integers and a 1 indicates little-endian integers. The CharRep field indicates what character format is used: 0 means ASCII and 1 means EBCDIC. Next, the FloatRep byte defines which floating-point representation is being used: 0 means IEEE 754, 1 means VAX, 2 means Cray, and 3 means IBM. The final 2 bytes are reserved for future use.

[3]DCE is defined by the Open Software Foundation (OSF), which is a consortium of companies that includes IBM, Digital, and Hewlett Packard.

Ex.1022 / Page 29 of 63
Intel Corporation and Lenovo (United States) Inc.

Note that in simple cases, such as arrays of integers, NDR does the same amount of work as XDR, and so it is able to achieve the same performance.

7.2 Data Compression

It is sometimes the case that application programs need to send more data in a timely fashion than the bandwidth of the network supports. For example, a video application might have a 10-Mbps video stream that it wants to transmit, but it has only a 1-Mbps network available to it. In cases like this, the data can first be *compressed* at the sender, then transmitted over the network, and finally *decompressed* at the receiver.

The field of data compression has a rich history, dating back to Shannon's pioneer work on information theory in the 1940s. If you think of information theory as the study of techniques for encoding data, then compression is about the efficiency of those encodings—how few bits are needed. In a nutshell, data compression is concerned with removing redundancy from the encoding.

In many ways, compression is inseparable from data encoding. That is, in thinking about how to encode a piece of data in a set of bits, we might just as well think about how to encode the data in the smallest set of bits possible. For example, if you have a block of data that is made up of the symbols A through Z, and if all of these symbols have an equal chance of occurring in the data block you are encoding, then encoding each symbol in 5 bits is the best you can do. If, however, the symbol R occurs 50% of the time, then it would be a good idea to use fewer bits to encode the R than any of the other symbols. In general, if you know the relative probability that each symbol will occur in the data, then you can assign a different number of bits to each possible symbol in a way that minimizes the number of bits it takes to encode a given block of data. This is the essential idea of *Huffman codes*, one of the important early developments in data compression.

For our purposes, there are two classes of compression algorithms. The first, called *lossless compression*, ensures that the data recovered from the compression/decompression process is exactly the same as the original data. A lossless compression algorithm is used to compress file data, such as executable code, text files, and numeric data, because programs that process such file data cannot tolerate mistakes in the data. In contrast, *lossy compression* does not promise that the data received is exactly the same as the data sent. This is because a lossy algorithm removes information that it cannot later restore. Hopefully, however, the lost information will not be missed by the receiver. Lossy algorithms are used to compress digital imagery, including video. This makes sense because such data often contains more information than the human eye can perceive, and for that matter, may already contains errors and imperfections that the human eye is able to compensate for. It is also the case that lossy algorithms typically achieve much better compression ratios than do their lossless counterparts; they can be as much as an order of magnitude better.

It might seem that compressing your data before sending it would always be a good idea, since the network would be able to deliver compressed data in less time than uncompressed

data. This is not necessarily the case, however. Compression/decompression algorithms often involve time-consuming computations. The question you have to ask is whether or not the time it takes to compress/decompress the data is worthwhile given such factors as the host's processor speed and the network bandwidth. Specifically, if B_c is the average bandwidth at which data can be pushed through the compressor and decompressor (in series), B_n is the network bandwidth (including network processing costs) for uncompressed data, r is the average compression ratio, and if we assume that all the data is compressed before any of it is transmitted, then the time taken to send x bytes of uncompressed data is x/B_n, whereas the time to compress it and send the compressed data is $x/B_c + x/(rB_n)$. Thus, compression is beneficial if

$$x/B_c + x/(rB_n) < x/B_n$$

which is equivalent to

$$B_c > r/(r-1) \times B_n$$

For example, for a compression ratio of 2, B_c would have to be greater than $2 \times B_n$ for compression to make sense.

7.2.1 Lossless Compression Algorithms

We begin by introducing three lossless compression algorithms. We do not describe these algorithms in much detail—we just give the essential idea—since it is the lossy algorithms used to compress image and video data that are of the greatest interest in today's network environment. We do comment, though, on how well these lossless algorithms work on digital imagery. Some of the ideas exploited by these lossless techniques show up again in later sections when we consider the lossy algorithms that are used to compress images.

Run Length Encoding

Run length encoding (RLE) is a compression technique with a brute-force simplicity. The idea is to replace consecutive occurrences of a given symbol with only one copy of the symbol, plus a count of how many times that symbol occurs; hence the name "run length." For example, the string AAABBCDDDD would be encoded as 3A2B1C4D.

RLE can be used to compress digital imagery by comparing adjacent pixel values and then encoding only the changes. For images that have large homogeneous regions, this technique is quite effective. For example, it is not uncommon that RLE can achieve compression ratios on the order of 8-to-1 for scanned text images. RLE works well on such files because they often contain a large amount of white space that can be removed. In fact, RLE is the key compression algorithm used to transmit faxes. However, for images with even a small degree of local variation, it is not uncommon for compression to actually increase the image byte size, since it takes two bytes to represent a single symbol when that symbol is not repeated.

Differential Pulse Code Modulation

Another simple lossless compression algorithm is Differential Pulse Code Modulation (DPCM). The idea here is to first output a reference symbol, and then for each symbol in the data, to output the difference between that symbol and the reference symbol. For example, using symbol A as the reference symbol, the string AAABBCDDDD would be encoded as A0001123333 since A is the same as the reference symbol, B has a difference of 1 from the reference symbol, and so on. Note that this simple example does not illustrate the real benefit of DPCM, which is that when the differences are small, they can be encoded with fewer bits than the symbol itself. In this example, the range of differences 0–3 can be represented with 2 bits each, rather than the 7 or 8 bits required by the full character. As soon as the difference becomes too large, a new reference symbol is selected.

DPCM works better than RLE for most digital imagery, since it takes advantage of the fact that adjacent pixels are usually similar. Due to this correlation, the dynamic range of the differences between the adjacent pixel values can be significantly less than the dynamic range of the original image, and this range can therefore be represented using fewer bits. Using DPCM, we have measured compression ratios of 1.5-to-1 on digital images.

Dictionary-Based Methods

The final lossless compression method we consider is the dictionary-based approach, of which the Lempel-Ziv (LZ) compression algorithm is the best known. The Unix `compress` command uses a variation of the LZ algorithm.

The idea of a dictionary-based compression algorithm is to build a dictionary (table) of variable-length strings (think of them as common phrases) that you expect to find in the data, and then to replace each of these strings when it appears in the data with the corresponding index to the dictionary. For example, instead of working with individual characters in text data, you could treat each word as a string and output the index in the dictionary for that word. To further elaborate on this example, the word "compression" has the index 4978 in one particular dictionary; it is the 4978th word in `/usr/share/dict/words`. To compress a body of text, each time the string "compression" appears, it would be replaced by 4978. Since this particular dictionary has just over 25,000 words in it, it would take 15 bits to encode the index, meaning that the string "compression" could be represented in 15 bits rather than the 77 bits required by 7-bit ASCII. This is a compression ratio of 5-to-1!

Of course, this leaves the question of where the dictionary comes from. One option is to define a static dictionary, preferably one that is tailored for the data being compressed. A more general solution, and the one used by LZ compression, is to adaptively define the dictionary based on the contents of the data being compressed. In this case, however, the dictionary constructed during compression has to be sent along with the data so that the decompression half of the algorithm can do its job. Exactly how you build an adaptive dictionary has been a subject of extensive research; we discuss important papers on the subject at the end of this chapter.

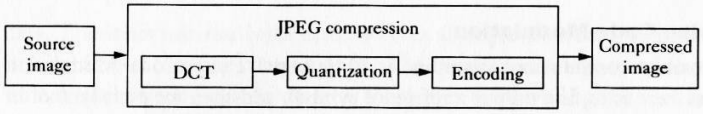


Figure 7.11 Block diagram of JPEG compression.

A variation of the LZ algorithm is used to compress digital images in the Graphical Interchange Format (GIF). Before doing that, GIF first reduces 24-bit color images to 8-bit color images. This is done by identifying the colors used in the picture, of which there will typically be considerably fewer than 2^{24}, and then picking the 256 colors that most closely approximate the colors used in the picture. These colors are stored in a table, which can be indexed with an 8-bit number, and the value for each pixel is replaced by the appropriate index. Note that this is an example of lossy compression for any picture with more than 256 colors. GIF then runs an LZ variant over the result, treating common sequences of pixels as the strings that make up the dictionary. Using this approach, GIF is sometimes able to achieve compression ratios on the order of 10-to-1, but only when the image consists of a relatively small number of discrete colors. Images of natural scenes, which often include a more continuous spectrum of colors, cannot be compressed at this ratio using GIF. As another data point, we were able to get a 2-to-1 compression ratio when we applied the LZ-based Unix `compress` command to the *x*-kernel distribution.

7.2.2 Image Compression (JPEG)

Given the increase in the use of digital imagery in the past few years—this use was spawned by the invention of graphical displays, not high-speed networks—the need for compression algorithms designed for digital imagery data has grown more and more critical. In response to this need, the ISO has defined a digital image format known as JPEG, named after the Joint Photographic Experts Group that designed it. (The "Joint" in JPEG stands for a joint ISO/ITU effort.) This section describes the compression algorithm at the heart of JPEG. The next section then describes a related format—MPEG—that is used for video data.

Before describing JPEG compression, one point that needs to be made is that JPEG, as well as GIF and MPEG, is more than just a compression algorithm. They also define the *format* for image or video data, much the same way that XDR, NDL, and ASN.1 define the format for numeric and string data. However, this section concentrates on the compression aspects of these standards.

JPEG compression takes place in three phases, as illustrated in Figure 7.11. On the compression side, the image is fed through these three phases one 8×8 block at a time. The first phase applies the Discrete Cosine Transform (DCT) to the block. If you think of the image as a signal in the spatial domain, then DCT transforms this signal into an equivalent signal in the *spatial frequency* domain. After the DCT, the second phase applies a quantization

to the resulting signal, and in so doing, loses the least significant bits of that signal. The third phase encodes the final result, but in so doing, adds an element of lossless compression to the lossy compression achieved by the first two phases. Decompression follows these same three phases, but in reverse order.

The following discussion describes each phase in more detail. It is simplified by considering only grayscale images; color images are discussed at the end of this section. In the case of grayscale images, each pixel in the image is given by an 8-bit value that indicates the grayness (brightness) of the pixel, where 0 equals white and 255 equals black.

DCT Phase

DCT is a transformation closely related to the Fast Fourier Transform (FFT). It takes an 8×8 matrix of pixel values as input and outputs an 8×8 matrix of frequency coefficients. You can think of the input matrix as a 64-point signal that is defined in two spatial dimensions (x and y); DCT breaks this signal into 64 spatial frequencies. To get an intuitive feel for spatial frequency, imagine yourself moving across a picture in, say, the x direction. You would see the value of each pixel varying as some function of x. If this value changes slowly with increasing x, then it has a low spatial frequency, and if it changes rapidly, it has a high spatial frequency. So the low frequencies correspond to the gross features of the picture, while the high frequencies correspond to fine detail. The idea behind the DCT is to separate the gross features, which are essential to viewing the image, from the fine detail, which is less essential and, in some cases, might be barely perceived by the eye.

DCT, along with its inverse which is performed during decompression, is defined by the following formulas:

$$DCT(i,j) = \frac{1}{\sqrt{2N}} C(i)C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cos\left[\frac{(2y+1)i\pi}{2N}\right]$$

$$pixel(x,y) = \frac{1}{\sqrt{2N}} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} C(i)C(j)DCT(i,j) \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cos\left[\frac{(2y+1)i\pi}{2N}\right]$$

$$C(x) = \frac{1}{\sqrt{2}} \quad \text{if } x = 0, \text{ else 1 if } x > 0$$

The first frequency coefficient, at location (0,0) in the output matrix, is called the *DC coefficient*. Intuitively, we can see that the DC coefficient is a measure of the average value of the 64 input pixels. The other 63 elements of the output matrix are called the *AC coefficients*. They add the higher-spatial-frequency information to this average value. Thus, as you go from the first frequency coefficient toward the 64th frequency coefficient, you are moving from low-frequency information to high-frequency information; from the broad strokes of the

image to finer and finer detail. These higher-frequency coefficients are increasingly unimportant to the perceived quality of the image. It is the second phase of JPEG that decides which portion of which coefficients to throw away.

Quantization Phase

The second phase of JPEG is where the compression becomes lossy; DCT does not itself lose information, it just transforms the image into a form that makes it easier to know what information to remove. (Although not lossy, per se, there is of course some loss of precision during the DCT phase due to the use of fixed-point arithmetic.) Quantization is easy to understand—it's simply a matter of dropping the insignificant bits of the frequency coefficients.

To see how the quantization phase works, imagine that you want to compress some whole numbers less than 100, e.g., 45, 98, 23, 66, and 7. If you decided that knowing these numbers truncated to the nearest multiple of 10 is sufficient for your purposes, then you could divide each number by the quantum 10 using integer arithmetic, yielding 4, 9, 2, 6, and 0. These numbers can each be encoded in 4 bits rather than the 7 bits needed to encode the original numbers.

Rather than using the same quantum for all 64 coefficients, JPEG uses a quantization table that gives the quantum to use for each of the coefficients, as specified in the formula given below. You can think of this table (Quantum) as a parameter that can be set to control how much information is lost, and correspondingly, how much compression is achieved. In practice, the JPEG standard specifies a set of quantization tables that have proven effective in compressing digital images; an example quantization table is given in Table 7.1. In tables like this one, the low coefficients have a quantum close to 1 (meaning that little low-frequency information is lost) and the high coefficients have larger values (meaning that more high-frequency information is lost). Notice that as a result of such quantization tables, many of the high-frequency coefficients end up being set to 0 after quantization, making them ripe for further compression in the third phase.

The basic quantization equation is

$$\mathsf{QuantizedValue}(i,j) = \mathsf{IntegerRound}(DCT(i,j)/\mathsf{Quantum}(i,j))$$

where

$$\mathsf{IntegerRound}(x) = \begin{cases} \lfloor x + 0.5 \rfloor & \text{if } x \geq 0 \\ \lfloor x - 0.5 \rfloor & \text{if } x < 0 \end{cases}$$

Decompression is then simply defined as

$$DCT(i,j) = \mathsf{QuantizedValue}(i,j) \times \mathsf{Quantum}(i,j)$$

For example, if the DC coefficient (i.e., DCT(0,0)) for a particular block was equal to 25, then the quantization of this value using Table 7.1 would result in $\lfloor 25/3 + 0.5 \rfloor = 8$. During decompression, this coefficient would then be restored as $8 \times 3 = 24$.

$$\text{Quantum} = \begin{bmatrix} 3 & 5 & 7 & 9 & 11 & 13 & 15 & 17 \\ 5 & 7 & 9 & 11 & 13 & 15 & 17 & 19 \\ 7 & 9 & 11 & 13 & 15 & 17 & 19 & 21 \\ 9 & 11 & 13 & 15 & 17 & 19 & 21 & 23 \\ 11 & 13 & 15 & 17 & 19 & 21 & 23 & 25 \\ 13 & 15 & 17 & 19 & 21 & 23 & 25 & 27 \\ 15 & 17 & 19 & 21 & 23 & 25 & 27 & 29 \\ 17 & 19 & 21 & 23 & 25 & 27 & 29 & 31 \end{bmatrix}$$

Table 7.1 Example JPEG quantization table.

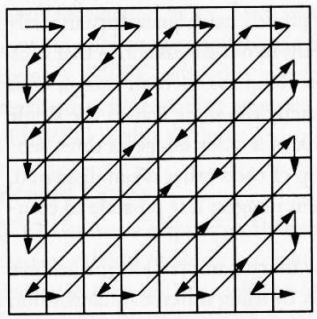


Figure 7.12 Zigzag traversal of quantized frequency coefficients.

Encoding Phase

The final phase of JPEG encodes the quantized frequency coefficients in a compact form. This results in additional compression, but this compression is lossless. Starting with the DC coefficient in position (0,0), the coefficients are processed in the zigzag sequence shown in Figure 7.12. Along this zigzag, a form of run length encoding is used—RLE is applied to only the 0 coefficients, which is significant because many of the later coefficients are 0. The individual coefficient values are then encoded using a Huffman code. (The JPEG standard allows the implementer to use an arithmetic coding instead of the Huffman code.)

In addition, because the DC coefficient contains a large percentage of the information about the 8×8 block from the source image, and images typically change slowly from block

to block, each DC coefficient is encoded as the difference from the previous DC coefficient. This is not unlike the strategy used in DPCM, described in Section 7.2.1.

Color Images

The preceding discussion assumed that each pixel was given by a single grayscale value. In the case of a color image, there are many different representations for each pixel to choose from. One representation, called RGB, represents each pixel with three color components: red, green, and blue. RGB is the representation of color typically supported by graphical input and output devices. Another representation, called YUV, also has three components: one luminance (Y) and two chrominance (U and V). Just like RGB, YUV is a three-dimensional coordinate system. However, compared to RGB, its coordinates are rotated to better match the human visual system. This is advantageous because the human visual system is not uniformly sensitive to colors. For example, we can distinguish the luminance (brightness) of a pixel much better than its hue (color).

Exactly why the three components in each of the two representations can be combined to produce acceptable color is an interesting question. The simple answer is that two-coordinate color systems have been defined, but they have proven inadequate for faithfully reproducing colors as perceived by humans. What is important to our discussion is that each pixel in a color image is given by three separate values. To compress such an image, each of these three components is processed independently in exactly the same way as the single grayscale value was processed. In other words, you can think of a color image as three separate images, where these separate images are overlaid on top of each other when displayed. Note that in general, JPEG is not limited to three-component images; it is possible to compress a multispectral image using JPEG.

JPEG includes a number of variations that control how much compression you achieve versus the fidelity of the image. This can be done, for example, by using different quantization tables. These variations, plus the fact that different images have different characteristics, make it impossible to say with any precision the compression ratios that can be achieved with JPEG. The widely accepted generalization, however, is that JPEG is able to compress 24-bit color images by a ratio of roughly 30-to-1: the image can first be compressed by a factor of three by reducing the 24 bits of color to 8 bits of color (as described for GIF) and then by another factor of 10 by using the algorithm described in this section.

7.2.3 Video Compression (MPEG)

We now turn our attention to the MPEG format, which is named after the Moving Pictures Expert Group that defined it. To a first approximation, a moving picture (i.e., video) is simply a succession of still images (frames) displayed at some video rate. Each of these frames can be compressed using the same DCT-based technique used in JPEG. Stopping at this point would be a mistake, however, because it fails to remove the frame-to-frame redundancy present in a video sequence. For example, two successive frames of video will contain almost identical information if there is not much motion in the scene, so it would be unnecessary to send the

Ex.1022 / Page 37 of 63
Intel Corporation and Lenovo (United States) Inc.

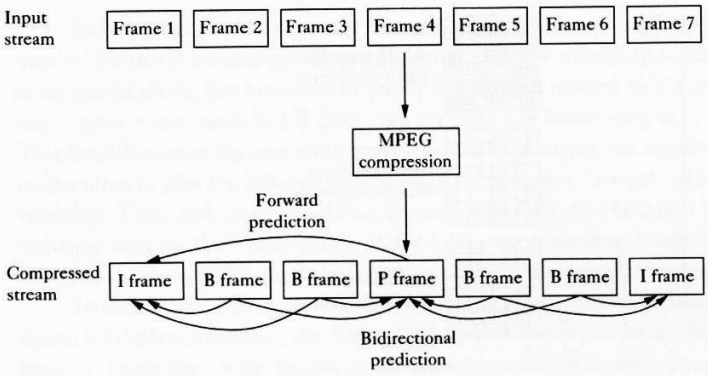


Figure 7.13 Sequence of I, P, and B frames generated by MPEG.

same information twice. Even when there is motion, there may be plenty of redundancy since a moving object may not change from one frame to the next; in some cases, only its position changes. MPEG takes this interframe redundancy into consideration. MPEG also defines a mechanism for encoding an audio signal with the video, but we consider only the video aspect of MPEG in this section.

Frame Types

MPEG takes a sequence of video frames as input and compresses them into three types of frames, called *I frames* (intrapicture), *P frames* (predicted picture), and *B frames* (bidirectional predicted picture). Each frame of input is compressed into one of these three frame types. I frames can be thought of as reference frames; they are self-contained, depending on neither earlier frames nor later frames. To a first approximation, an I frame is simply the JPEG compressed version of the corresponding frame in the video source. P and B frames are not self-contained; they specify relative differences from some reference frame. More specifically, a P frame specifies the differences from the previous I frame, while a B frame gives an interpolation between the previous and subsequent I or P frames.

Figure 7.13 illustrates a sequence of seven video frames that, after being compressed by MPEG, result in a sequence of I, P, and B frames. The two I frames stand alone; each can be decompressed at the receiver independently of any other frames. The P frame depends on the preceding I frame; it can be decompressed at the receiver only if the preceding I frame also arrives. Each of the B frames depend on both the preceding I or P frame and the subsequent I or P frame. Both of these reference frames must arrive at the receiver before MPEG can decompress the B frame to reproduce the original video frame.

Note that because each B frame depends on a later frame in the sequence, the compressed frames are not transmitted in sequential order. Instead, the sequence I B B P B B I

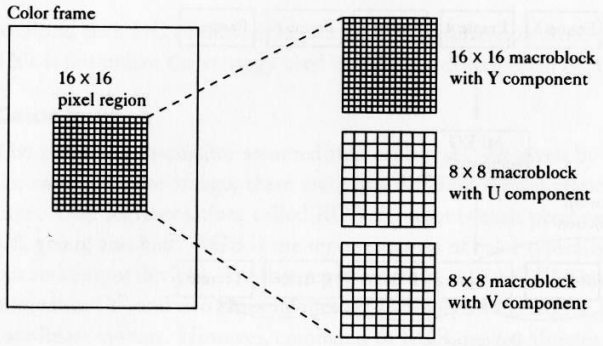


Figure 7.14 Each frame as a collection of macroblocks.

shown in Figure 7.13 is transmitted as I P B B I B B. Also, MPEG does not define the ratio of I frames to P and B frames; this ratio may vary depending on the required compression and picture quality. For example, it is permissible to transmit only I frames. This would be similar to using JPEG to compress the video.

In contrast to the preceding discussion of JPEG, the following focuses on the *decoding* of an MPEG stream. It is a little easier to describe, and it is the operation that is more often implemented in networking systems today, since MPEG coding is so expensive that it is normally done offline, i.e., not in real time. For example, in a video-on-demand system, the video would be encoded and stored on disk ahead of time. When a viewer wanted to watch the video, the MPEG stream would then be transmitted to the viewer's machine which would decode and display the stream in real time.

Let's look more closely at the three frame types. As mentioned above, I frames are approximately equal to the JPEG compressed version of the source frame. The main difference is that MPEG works in units of 16×16 *macroblocks*. For a color video represented in YUV, the U and V components in each macroblock are "down sampled" into an 8×8 block. That is, each 2×2 subblock in the macroblock is given by one U value and one V value—the average of the four pixel values. The subblock still has four Y values. This can be done because the U and V components can be transmitted less accurately without visibly disturbing the image, since humans are less sensitive to color than they are to brightness. The relationship between a frame and the corresponding macroblocks is given in Figure 7.14.

The P and B frames are also processed in units of macroblocks. Intuitively, we can see that the information they carry for each macroblock captures the motion in the video, that is, it shows in what direction and how far the macroblock moved relative to the reference frame(s). The following describes how a B frame is used to reconstruct a frame during decompression; P frames are handled in a similar manner, except that they depend on only one reference frame instead of two.

Before getting to the details of how a B frame is decompressed, we first note that each macroblock in a B frame is not necessarily defined relative to both an earlier and a later frame, as suggested above, but may instead simply be specified relative to just one or the other. In fact, a given macroblock in a B frame can use the same intracoding as is used in an I frame. This flexibility exists because if the motion picture is changing too rapidly, then it sometimes makes sense to give the intrapicture encoding rather than a forward- or backward-predicted encoding. Thus, each macroblock in a B frame includes a type field that indicates which encoding is used for that macroblock. In the following discussion, however, we consider only the general case in which the macroblock uses bidirectional predictive encoding.

In such a case, each macroblock in a B frame is represented with a 4-tuple: (1) a coordinate for the macroblock in the frame, (2) a motion vector relative to the previous reference frame, (3) a motion vector relative to the subsequent reference frame, and (4) a delta (δ) for each pixel in the macroblock (i.e., how much each pixel has changed relative to the two reference pixels). For each pixel in the macroblock, the first task is to find the corresponding reference pixel in the past and future reference frames. This is done using the two motion vectors associated with the macroblock. Then, the delta for the pixel is added to the average of these two reference pixels. Stated more precisely, if we let F_p and F_f denote the past and future reference frames, respectively, and the past/future motion vectors are given by (x_p, y_p) and (x_f, y_f), then the pixel at coordinate (x, y) in the current frame (denoted F_c) is computed as

$$F_c(x, y) = (F_p(x + x_p, y + y_p) + F_f(x + x_f, y + y_f))/2 + \delta(x, y)$$

where δ is the delta for the pixel as specified in the B frame. These deltas are encoded in the same way as pixels in I frames. That is, they are run through DCT and then quantized. Since the deltas are typically small, most of the DCT coefficients are 0 after quantization, hence they can be effectively compressed.

It should be fairly clear from the preceding discussion how encoding would be performed, with one exception. When generating a B or P frame during compression, MPEG must decide where to place the macroblocks. Recall that each macroblock in a P frame, for example, is defined relative to a macroblock in an I frame, but that the macroblock in the P frame need not be in the same part of the frame as the corresponding macroblock in the I frame—the difference in position is given by the motion vector. You would like to pick a motion vector that makes the macroblock in the P frame as similar as possible to the corresponding macroblock in the I frame, so that the deltas for that macroblock can be as small as possible. This means that you need to figure out where objects in the picture moved from one frame to the next. This is the problem of *motion estimation*, and several techniques (heuristics) for solving this problem are known. (We discuss papers that consider this problem at the end of this chapter.) The difficulty of this problem is one of the reasons that MPEG encoding takes longer than decoding on equivalent hardware. MPEG does not specify any particular technique; it only defines the format for encoding this information in B and P frames and the algorithm for reconstructing the pixel during decompression, as given above.

Effectiveness and Performance

MPEG typically achieves a compression ratio of 90-to-1, although ratios as high as 150-to-1 are not unheard of. In terms of the individual frame types, one can expect a compression ratio of approximately 30-to-1 for the I frames (this is consistent with the ratios achieved using JPEG when 24-bit color is first reduced to 8-bit color), while P and B frame compression ratios are typically three to five times smaller than the rates for the I frame. Without first reducing the 24 bits of color to 8 bits, the achievable compression with MPEG is typically between 30-to-1 and 50-to-1.

MPEG involves an expensive computation. On the compression side, it is typically done offline, which is not a problem for preparing movies for a video-on-demand service. Video can be compressed in real time using hardware, but such devices are fairly expensive today. On the decompression side, low-cost MPEG video boards are available, but they do little more than YUV color lookup, which fortunately is the most expensive step. Most of the actual MPEG decoding is done in software. Until recently, processors have not been fast enough to keep pace with 30-frames-per-second video rates when decoding MPEG streams purely in software. With a 175-MHz RISC architecture available in 1994, however, it is possible to decompress MPEG fast enough to keep up with a 320×240 video stream running at 30 frames per second.

7.3 Security

Computer networks are typically a shared resource used by many applications for many different purposes. Sometimes the data transmitted between application processes is confidential, and the applications would prefer that others not be able to read it. For example, when purchasing a product over the World Wide Web, users sometimes transmit their credit card numbers over the network. This is a dangerous thing to do since it is easy for someone to eavesdrop on the network and read all the packets that fly by. Therefore, a third transformation made to data sent over an end-to-end channel is to *encrypt* it, with the goal of keeping anyone who is eavesdropping on the channel from being able to read the contents of the message.

The idea of encryption is simple enough: the sender applies an *encryption* function to the original *plaintext* message, the resulting *ciphertext* message is sent over the network, and the receiver applies a reverse function (called *decryption*) to recover the original plaintext. The encryption/decryption process generally depends on a secret *key* shared between the sender and the receiver. When a suitable combination of a key and an encryption algorithm is used, it is sufficiently difficult for an eavesdropper to break the ciphertext and the sender and receiver can rest assured that their communication is secure.

This familiar use of cryptography is designed to ensure privacy—preventing the unauthorized release of information. Privacy, however, is not the only service that cryptography provides. It can also be used to support other equally important services, including authentication (verifying the identity of the remote participant) and message integrity (making sure that the message has not been altered). This section first introduces the basic idea of

Ex.1022 / Page 41 of 63
Intel Corporation and Lenovo (United States) Inc.

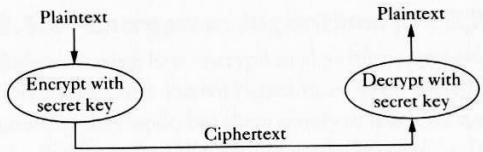


Figure 7.15 Secret key encryption.

cryptography—including a description of the two most common encryption algorithms, the Data Encryption Standard (DES) and Rivest, Shamir, and Adleman (RSA)—and then shows how these algorithms can be used to provide authentication and integrity services.

One thing to keep in mind while reading this section is that the various algorithms and protocols for privacy, authentication, and integrity are being described in isolation. In practice, constructing a secure system requires an intricate combination of just the right set of protocols and algorithms. This is a challenging task because each protocol is vulnerable to a different set of attacks. To make matters worse, determining when a security protocol is "good enough" is as much art and politics as science. A thorough analysis of these different attacks, and how you might build a complete system that minimizes the risk of compromise, is beyond the scope of this book.

7.3.1 Secret Key versus Public Key Cryptography

Broadly speaking, there are two types of cryptographic algorithms: *secret key* algorithms and *public key* algorithms. Secret key algorithms are symmetric in the sense that both participants[4] in the communication share a single key. Figure 7.15 illustrates the use of secret key encryption to transmit data over an otherwise insecure channel. The Data Encryption Standard (DES) is the best-known example of a secret key encryption function.

In contrast to a pair of participants sharing a single secret key, *public key* cryptography involves each participant having a *private* key that is shared with no one else and a *public* key that is published so everyone knows it. To send a secure message to this participant, you encrypt the message using the widely known public key. The participant then decrypts the message using his or her private key. This scenario is depicted in Figure 7.16. RSA—named after its inventors, Rivest, Shamir, and Adleman—is the best-known public key encryption algorithm.

There is actually a third type of cryptography algorithm, called a *hash* or *message digest* function. Unlike the preceding two types of algorithms, cryptographic hash functions involve the use of no keys. Instead, the idea is to map a potentially large message into a small fixed-length number, analogous to the way a regular hash function maps values from a large space into values from a small space.

[4]We use the term *participant* for the parties involved in a secure communication since that is the term we have been using throughout the book to identify the two endpoints of a channel. In the security world, they are typically called *principals*.

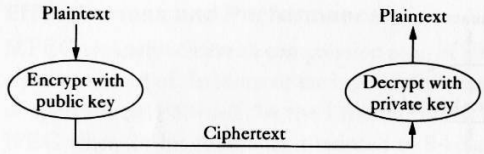


Figure 7.16 Public key encryption.

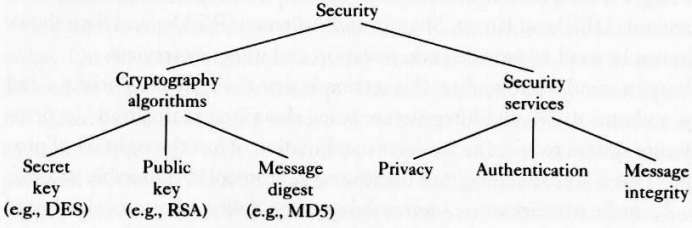


Figure 7.17 Taxonomy of network security.

The best way to think of a cryptographic hash function is that it computes a *cryptographic checksum* over a message. That is, just as a regular checksum protects the receiver from accidental changes to the message, a cryptographic checksum protects the receiver from malicious changes to the message. This is because all cryptographic hash algorithms are carefully selected to be one-way functions—given a cryptographic checksum for a message, it is virtually impossible to figure out what message produced that checksum. Said another way, it is not computationally feasible to find two messages that hash to the same cryptographic checksum. The relevance of this property is that if you are given a checksum for a message (along with the message) and you are able to compute exactly the same checksum for that message, then it is highly likely that this message produced the checksum you were given.

Although we will not describe it in detail, the most widely used cryptographic checksum algorithm is Message Digest version 5 (MD5). An important property of MD5, in addition to those properties outlined in the previous paragraph, is that it is much more efficient to compute than either DES or RSA. We will see the relevance of this fact later in this section.

To re-emphasize, cryptography algorithms like DES, RSA, and MD5 are just building blocks from which a secure system can be constructed. Figure 7.17 gives a simple taxonomy that illustrates this point. In looking at these services and building blocks, we should consider the following question: How did the participants get the various keys in the first place? This is the *key distribution* problem, one of the central problems in security, as we will see in the following subsections.

Ex.1022 / Page 43 of 63
Intel Corporation and Lenovo (United States) Inc.

7.3.2 Encryption Algorithms (DES, RSA)

Before showing how encryption algorithms are used to build secure systems, we first describe how the two best-known algorithms—DES and RSA—work. We will also give some insight into *why* they work, but there is only so much we can do on this front since the design principles that underlie DES are not public knowledge. In the case of RSA, a deep explanation for why it works would require a background in number theory that is beyond the scope of this book, but we can provide some intuition into the underlying principles. Before looking at either algorithm, however, let's step back and ask what we want from an encryption algorithm.

Requirements

The basic requirement for an encryption algorithm is that it be able to turn plaintext into ciphertext in such a way that only the intended recipient—the holder of the decryption key—can recover the plaintext. What this means is that the encryption method should be safe from attacks by people who do not hold the key. As a starting point, we should assume that the encryption algorithm itself is known and that only the key is kept secret. The reason for this assumption is that if you depend on the algorithm being kept secret, then you have to throw it out when you believe it is no longer secret. This means potentially frequent changes of algorithm, which is problematic since it takes a lot of work to develop a new algorithm. Also, one of the best ways to know that an algorithm is effective is to use it for a long time—if no one breaks it, it's probably secure. (Fortunately, there are plenty of people who will try to break algorithms and who will let it be widely known when they have succeeded, so no news is generally good news.) Thus, there is considerable risk in deploying a new algorithm. Therefore, our first requirement is that secrecy of the key, and not of the algorithm itself, is the only thing that is needed to ensure the privacy of the data.

It is important to realize that when someone receives a piece of ciphertext, they may have more information at their disposal than just the ciphertext itself. For example, they may know that the plaintext was written in English, which means that the letter e occurs more often in the plaintext that any other letter; the frequency of many other letters and common letter combinations can also be predicted. This information can greatly simplify the task of finding the key. Similarly, they may know something about the likely contents of the message; e.g., the word "login" is likely to occur at the start of a remote login session. This may enable a "known plaintext" attack, which has a much higher chance of success than a "ciphertext only" attack. Even better is a "chosen plaintext" attack, which may be enabled by feeding some information to the sender that you know the sender is likely to transmit—such things have happened in wartime, for example.

The best cryptographic algorithms, therefore, can prevent the attacker from deducing the key even when the individual knows both the plaintext and the ciphertext. One approach, the one taken in DES, is to make the algorithm so complicated that virtually none of the structure of the plaintext remains in the ciphertext. This leaves the attacker with no choice but to search the space of possible keys exhaustively. This can be made infeasible by choosing a suit-

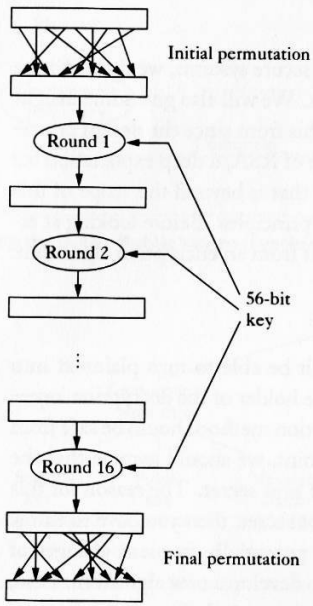


Figure 7.18 High-level outline of DES.

ably large key space and by making the operation of checking a key reasonably costly. As we will see, DES is now becoming only marginally secure on that basis.

DES

DES encrypts a 64-bit block of plaintext using a 64-bit key. The key actually contains only 56 usable bits—the last bit of each of the 8 bytes in the key is a parity bit for that byte. Also, messages larger than 64 bits can be encrypted using DES, as described below.

DES has three distinct phases: (1) the 64 bits in the block are permuted (shuffled), (2) 16 rounds of an identical operation are applied to the resulting data and the key, and (3) the inverse of the original permutation is applied to the result. This high-level outline of DES is depicted in Figure 7.18.

A table representing part of the initial permutation is shown in Table 7.2. The final permutation is the inverse (e.g., bit 40 would be permuted to bit position 1). It is generally agreed that these two permutations add nothing to the security of DES. Some security experts speculate that they were included to make the computation take longer, but it is just as likely that they are an artifact of the initial hardware implementation, involving some restriction of pin layout, for example.

Input Position	1	2	3	4	5	...	60	61	62	63	64
Output Position	40	8	48	16	56	...	9	49	17	57	25

Table 7.2 Initial (and final) DES permutation.

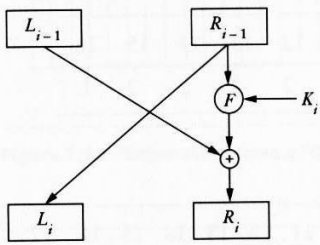


Figure 7.19 Manipulation at each round of DES.

During each round, the 64-bit block is broken into two 32-bit halves, and a different 48 bits are selected from the 56-bit key. If we denote the left and right halves of the block at round i as L_i and R_i, respectively, and the 48-bit key at round i as K_i, then these three pieces are combined during round i according to the following rule:

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$$

where F is a combiner function described below and $\oplus$ is the exclusive-OR (XOR) operator. Figure 7.19 illustrates the basic operation of each round. Note that L_0 and R_0 correspond to the left and right halves of the 64-bit block that results from the initial permutation, and that L_{16} and R_{16} are combined back together to form the 64-bit block to which the final inverse permutation is applied.

We now need to define function F and show how each K_i is derived from the 56-bit key. We start with the key. Initially, the 56-bit key is permuted according to Table 7.3. Note that every eighth bit is ignored (i.e., bit 64 is missing from the table), reducing the key from 64 bits to 56 bits. Then for each round, the current 56 bits are divided into two 28-bit halves and each half is independently shifted left either one or two digits, depending on the round. The number of bits shifted in each round is given in Table 7.4. The 56 bits that result from this shift are used as both input for the next round (i.e., the preceding shift is repeated) and to select the 48 bits that make up the key for the current round. Table 7.5 shows how 48 of the 56 bits are selected; note that they are simultaneously selected and permuted. For example, the bit in position 9 is not selected because it is not in the table.

Function F combines the resulting 48-bit key for round i (K_i) with the right half of the data block after round $i-1$ (R_{i-1}), as follows. To simplify our notation, we refer to K_i and

Input Position	1	2	3	4	5	...	59	60	61	62	63
Output Position	8	16	24	56	52	...	17	25	45	37	29

Table 7.3 DES key permutation.

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Shift	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Table 7.4 DES key shifts per round.

Input Position	1	2	3	4	5	6	7	8	10	11	12	13	14	15	16	17
Output Position	5	24	7	16	6	10	20	18	12	3	15	23	1	9	19	2

Input Position	19	20	21	23	24	26	27	28	29	30	31	32	33	34	36	37
Output Position	14	22	11	13	4	17	21	8	47	31	27	48	35	41	46	28

Input Position	39	40	41	42	44	45	46	47	48	49	50	51	52	53	55	56
Output Position	39	32	25	44	37	34	43	29	36	38	45	33	26	42	30	40

Table 7.5 DES compression permutation.

R_{i-1} as K and R, respectively. First, function F expands R from 32 bits into 48 bits so that it can be combined with the 48-bit K. It does this by breaking R into eight 4-bit chunks and expanding each chunk into 6 bits by stealing the rightmost and leftmost bit from the left and right adjacent 4-bit chunks, respectively. This expansion is illustrated in Figure 7.20, where R is treated as circular in the sense that the first and last chunks get their extra bit from each other.

Next, the 48-bit K is divided into eight 6-bit chunks, and each chunk is XORed with the corresponding chunk that resulted from the previous expansion of R. Finally, each resulting 6-bit value is fed through something called a *substitution box* (S box), which reduces each 6-bit chunk back into 4 bits. There are actually eight different S boxes, one for each of the 6-bit chunks. You can think of an S box as just performing a many-to-one mapping from 6-bit numbers to 4-bit numbers. Table 7.6 gives part of the S box function for the first chunk. We are now done with round i.

Notice that the preceding description does not distinguish between encryption and decryption. One of the nice features of DES is that both sides of the algorithm work exactly

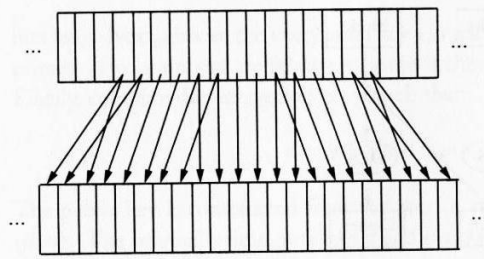


Figure 7.20 Expansion phase of DES.

Input	000000	000001	000010	000011	000100	000101	...
Output	1110	0100	1101	0001	0010	1111	...

Input	...	111010	111011	111100	111101	111110	111111
Output	...	0011	1110	1010	0000	0110	1101

Table 7.6 Example DES S box (bits 1-6).

the same. The only difference is that the keys are applied in the reverse order, i.e., K_{16}, $K_{15}, \ldots, K_1$.

Also keep in mind that the preceding discussion is limited to a single 64-bit data block. To encrypt a longer message using DES, a technique known as *cipher block chaining* (CBC) is typically used. The idea of CBC is simple: the ciphertext for block i is XORed with the plaintext for block $i + 1$ before running it through DES. An *initialization vector* (IV) is used in lieu of the nonexistent ciphertext for block 0. This vector IV, which is a random number generated by the sender, is sent along with the message so that the first block of plaintext can be retrieved. CBC on the encryption side is shown in Figure 7.21 for a 256-bit (four-block) message. Decryption works in the expected way since XOR is its own inverse, with the process starting with the last block and moving toward the front of the message.

We conclude by noting that there is no mathematical proof that DES is secure. What security it achieves it does through the application of two techniques: confusion and diffusion. (Having just plowed through the algorithm, you should now have a deep understanding of these two techniques.) What we can say is that the only known way to break DES is to exhaustively search all possible 2^{56} keys, although on average you would expect to have to search only half of the key space, or $2^{55} = 3.6 \times 10^{16}$ keys. On a 175-MHz DEC Alpha workstation, it is possible to do one encryption in 4 μs, meaning that it would take 1.4×10^{17} μs to break

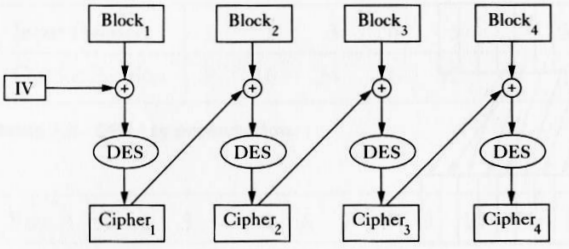


Figure 7.21 Cipher block chaining (CBC) for large messages.

a key (approximately 4500 years). While that may seems like a long time, keep in mind that searching a key space is a *highly* parallel task, meaning that if you could throw 9000 Alphas at the job, it would take only six months to break a key.

This amount of time is considered borderline-secure in many circles, especially considering that processor speeds are doubling every 18 months. For this reason, many applications are starting to use triple-DES, that is, encrypting the data three times. This can be done with three separate keys, or with two keys: the first is used, then the second, and finally the first key is used again.

RSA

RSA is a much different algorithm, not only because it involves different keys for encryption (public key) and decryption (private key), but also because it is grounded in number theory. In fact, the essential aspect of RSA comes down to how these two keys are selected. The act of encrypting or decrypting a message is expressed as a simple function, although this function requires enormous computational power. In particular, RSA commonly uses a key length of 512 bits, making it much more expensive to compute than DES; we discuss this more below.

The first task is to generate a public and private key. To do this, choose two large prime numbers p and q, and multiply them together to get n. Both p and q should be roughly 256

Breaking RSA

In 1977, a challenge was issued to break a 129-digit (430-bit) message that was encrypted using RSA. It was believed that the code was impregnable, requiring 40 quadrillion years of computation using the currently known algorithms for factoring large numbers. In April of 1994, a mere 17 years later, four scientists reported that they had broken the code. The hidden message was

THE MAGIC WORDS ARE
SQUEAMISH OSSIFRAGE

The task was accomplished using a factoring method that requires approximately 5000 MIP-years. This was done over an eight-month period of time by dividing the

bits long. Next, choose the encryption key e, such that e and $(p-1) \times (q-1)$ are relatively prime. (Two numbers are relatively prime if they have no common factor greater than 1.) Finally, compute the decryption key d such that

$$d = e^{-1} \bmod ((p-1) \times (q-1))$$

The public key is constructed from the pair $\langle e, n \rangle$ and the private key is given by the pair $\langle d, n \rangle$. The original prime numbers p and q are no longer needed. They can be discarded, but they must not be disclosed.

Given these two keys, encryption is defined by the following formula:

$$c = m^e \bmod n$$

and decryption is defined by

$$m = c^d \bmod n$$

where m is the plaintext message and c is the resulting ciphertext. Note that m must be less than n, which means that it can be no more than the 512 bits long. A larger message is simply treated as the concatenation of multiple 512-bit blocks.

Notice that when two participants want to encrypt data they are sending to each other using a public key algorithm like RSA, a pair of public/private key pairs is required. It doesn't work to encrypt with your private key and let the other side decrypt with the public key because everyone has access to the public key and so could decrypt the message. In other words, participant A encrypts data it sends to participant B using B's public key and B uses its private key to decrypt this data, while B encrypts data it sends to A using A's public key and A decrypts this message using its private key. Observe that A cannot decrypt a message that it has sent to B; only B has the requisite private key.

RSA security comes from the premise that factoring large numbers is a computationally expensive proposition. In particular, if you could factor n, you could recover p and q, which would compromise d. The speed at which large numbers can be factored is a function

problem into smaller pieces and shipping these pieces, using email, to computers all over the world.

Keep in mind that it doesn't always take 5000 MIP-years to break a key, especially when the key is poorly chosen. For example, a security hole was recently exposed in a WWW browser that used RSA to encrypt credit card numbers that were being sent over the Internet. The problem was that the system used a highly predictable method (a combination of process ID plus time of day) to generate a random number that was, in turn, used to generate a private and public key. Such keys are easily broken.

of both the available processor speed and the factoring algorithm being used. It is estimated that 512-bit numbers will be factorable in the next few years, and in fact, people are already starting to use 768- and 1024-bit keys. Keep in mind that while we are concentrating on the security of data as it moves through the network—i.e., the data is sometimes vulnerable for only a short period of time—in general, security people have to consider the vulnerability of data that needs to be stored in archives for tens of years.

Implementation and Performance

DES is two to three orders of magnitude faster than RSA, depending on whether the implementations are done in hardware or software. When implemented in software, we have measured DES to process data at 36 Mbps and RSA to process data at only 1 Kbps on a 175-MHz DEC Alpha workstation. For comparison, a software implementation of MD5 can process data at a rate of 80 Mbps on the same workstation. When implemented in hardware, that is, by custom VLSI chips, it has been reported that DES can achieve rates approaching 1 Gbps and RSA can achieve a whopping 64 Kbps.

Perhaps surprisingly, DES is the more likely of the two algorithms to be implemented in hardware on a given computer. This is because even when implemented in hardware, RSA is still too slow to be of any practical use in encrypting data messages. Instead, RSA is typically used to encrypt secret keys that are passed between two participants. These participants then use DES, possibly implemented in hardware, to secure their communication using this key. Because the secret key would only be used for as long as these two participants want to communicate, it is called a *session* key. We discuss this more in following subsections.

7.3.3 Authentication Protocols (Kerberos)

Before two participants are likely to establish a secure channel between themselves—i.e., use DES or RSA to encrypt messages they exchange—they must each first establish that the other participant is who he or she claims to be. This is the problem of authentication. If you think about authentication in the context of a client/server relationship, say a remote file system, then it is understandable that the server would want to establish the identity of the client: If the client is going to be allowed to modify or delete John's file, then the server is obligated to make sure that the client is, in fact, John. It is also the case, however, that the client often wants to verify the identity of the server. After all, you would not want to start writing sensitive data to what you thought was a file server, only to later discover that it was an imposter process.

This section describes three common protocols for implementing authentication. The first two use secret key cryptography (e.g., DES), while the third uses public key cryptography (e.g., RSA). Note that it is often during the process of authentication that the two participants establish the session key that is going to be used to ensure privacy during subsequent communication. The following includes a discussion of how this process gets bootstrapped.

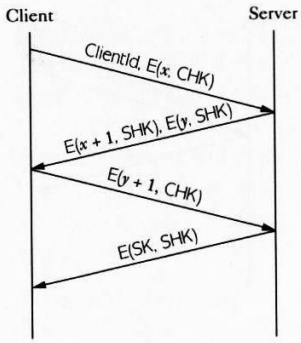


Figure 7.22 Three-way handshake protocol for authentication.

Simple Three-Way Handshake

A simple authentication protocol is possible when the two participants who want to authenticate each other—think of them as a client and a server—already share a secret key. This situation is analogous to a user (the client) having an account on a computer system (the server), where both client and server know the password for the account.

The client and server authenticate each other using a simple three-way handshake protocol similar to the one described in Section 6.2.3. In the following, we use E(m,k) to denote the encryption of message m with key k and D(m,k) to denote the decryption of message m with key k.

As illustrated in Figure 7.22, the client first selects a random number x and encrypts it using its secret key, which we denote as CHK (client handshake key). The client then sends E(x, CHK), along with an identifier (ClientId) for itself to the server. The server uses the key it thinks corresponds to client ClientId (call it SHK for server handshake key) to decrypt the random number. The server adds 1 to the number it recovers, and sends the result back to the client. It also sends back a random number y that has been encrypted with SHK. Next, the client decrypts the first half of this message and if the result is 1 more than the random number x that it sent to the server, it knows that the server possesses its secret key. At this point, the client has authenticated the server. The client also decrypts the random number the server sent it (this should yield y), encrypts this number plus 1, and sends the result to the server. If the server is able to recover $y + 1$ then it knows the client is legitimate.

After the third message, each side has authenticated itself to the other. The fourth message in Figure 7.22 corresponds to the server sending the client a session key (SK), encrypted using SHK (which is equal to CHK). Typically, the client and server then use SK to encrypt any future data they send to each other. The advantage of using a session key is that it means that

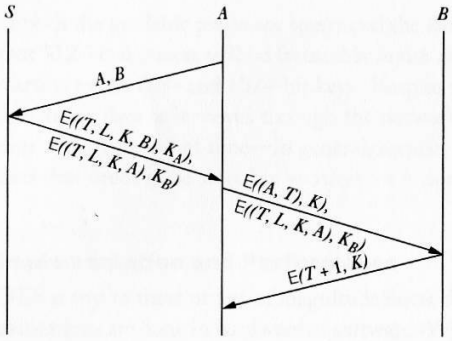


Figure 7.23 Third-party authentication in Kerberos.

the permanent secret key is only used for a small number of messages, making it harder for an attacker to gather data that might be used to determine the key.

This only begs the question of where the client and server handshake keys came from in the first place. One possibility is that they correspond to a password that a user entered; the ClientId could be the login identifier in this situation. Because a user-selected password might not make a suitable secret key, a transformation is often performed to turn it into a legitimate 56-bit DES key, for example.

Trusted Third Party

A more likely scenario is that the two participants know nothing about each other, but both trust a third party. This third party is sometimes called an *authentication server*, and it uses a protocol to help the two participants authenticate each other. There are actually many different variations of this protocol. The one we describe is the one used in Kerberos, a TCP/IP-based security system developed at MIT.

In the following, we denote the two participants who want to authenticate each other as A and B, and we call the trusted authentication server S. The Kerberos protocol assumes that A and B each share a secret key with S; we denote these two keys as K_A and K_B, respectively. As before, E(m,k) denotes message m encrypted with key k.

As illustrated in Figure 7.23, participant A first sends a message to server S that identifies both itself and B. The server then generates a timestamp T, a lifetime L, and a new session key K. Timestamp T is going to serve much the same purpose as the random number in the simple three-way handshake protocol given above, plus it is used in conjunction with L to limit the amount of time that session key K is valid. Participants A and B will have to go back to server S to get a new session key when this time expires. The idea here is to limit the vulnerability of any one session key.

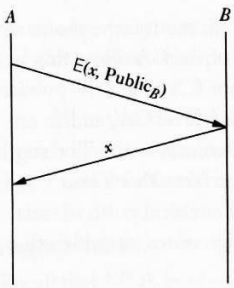


Figure 7.24 Public key authentication.

Server S then replies to A with a two-part message. The first part encrypts the three values T, L, and K, along with the identifier for participant B, using the key that the server shares with A. The second part encrypts the three values T, L, and K, along with participant A's identifier, but this time using the key that the server shares with B. Clearly, when A receives this message, it will be able to decrypt the first part but not the second part. A simply passes this second part on to B, along with the encryption of A and T using the new session key K. (A was able to recover T and K by decrypting the first part of the message it got from S.) Finally, B decrypts the part of the message from A that was originally encrypted by S, and in so doing, recovers T, K, and A. It uses K to decrypt the half of the message encrypted by A, and upon seeing that A and T are consistent in the two halves of the message, replies with a message that encrypts $T + 1$ using the new session key K.

A and B can now communicate with each other using the shared secret session key K to ensure privacy.

Public Key Authentication

Our final authentication protocol uses public key cryptography; e.g., RSA. The public key protocol is a useful one because the two sides need not share a secret key; they only need to know the other side's public key. As shown in Figure 7.24, participant A encrypts a random number x using participant B's public key, and B proves it knows the corresponding private key by decrypting the message and sending x back to A. A could authenticate itself to B in exactly the same way.

As to the question of where a participant learns the other's public key, it is tempting to believe that individual participants should be allowed to post their public keys on a bulletin board. This approach does not work, however, because it would be possible for participant A to post its own public key and claim that it is the key for participant B. This would allow A to masquerade as B. Instead, public keys must be retrieved from a trusted source, typically known as a *certificate authority* (CA). That is, you go to the trusted CA to get some other participant's public key, and you have to prove through some external mechanism that you are

who you say you are when you register your public key with the CA. This reduces the problem to one of every participant having to know just one public key—that of the CA. Note that in general, it is possible to build a hierarchy of CAs: you start at the root CA to get the public key for a second-level CA, which gives you the public key for the third-level CA, and so on, until you learn the public key for the participant you want to communicate with. The root CA would be published in some other trusted source, for example, the *New York Times*.

An alternative to a tree-structured hierarchy is to build a nonhierarchical mesh of trust. We defer discussion of this approach until we have discussed digital signatures, on which this approach depends.

7.3.4 Message Integrity Protocols

Sometimes two communicating participants do not care whether an eavesdropper is able to read the messages they are sending to each other, but they are worried about the possibility of an imposter sending messages that claims to be from one of them. That is, the participants want to ensure the integrity of their messages.

One way to ensure the integrity of a message is to encrypt it using DES with cipher block chaining, and then to use the CBC *residue* (the last block output by the CBC process) as a *message integrity code* (MIC). (For the CBC example given in Figure 7.21, cipher_4 is the CBC residue.) The plaintext message plus the MIC would be transmitted to the receiver, with the MIC acting as a sort of checksum—if the receiver could not reproduce the attached MIC using the secret key it shares with the sender, then the message was either not sent by the sender, or it was modified since it was transmitted. Note that you would not want to use DES with CBC to both encrypt the message for privacy and to generate the MIC for integrity, because you would simply end up transmitting the CBC-encrypted message with the last block repeated. Thus, anyone who wanted to tamper with the CBC-encrypted message could take the value of the final block they wanted to send, and send it twice.

This section looks at three alternatives for ensuring message integrity. The first uses RSA to produce a digital signature. RSA used on its own tends to be slow, but it can be used in combination with MD5 to yield a much more efficient technique. The second and third approaches use MD5 in conjunction with RSA to guarantee message integrity.

Digital Signature Using RSA

A *digital signature* is a special case of a message integrity code, where the code can only have been generated by one participant. The easiest digital signature algorithm to understand is an RSA signature, which works in the obvious way—since a given participant is the only one that knows its own private key, the participant uses this key to produce the signature. Any other participant can verify this signature using the corresponding public key. In other words, to sign a message, you encrypt it using your private key, and to verify a signature, you decrypt it using the public key of the purported sender. Clearly, this means that producing an RSA signature is as slow as RSA, which we have already seen is two or three orders of magnitude slower than DES. Observe that the use of keys is exactly reversed relative to their use for privacy:

the sender encrypts with the sender's private key rather than with the receiver's public key, and the receiver decrypts with the sender's public key rather than with the receiver's private key.

Note that the National Institute for Standards and Technology (NIST) has proposed a digital signature standard known as DSS that is similar to the approach just described, except that it uses an alternative algorithm, called El Gamel, instead of RSA.

Keyed MD5

Recall that MD5 produces a cryptographic checksum for a message. This checksum does not depend on a secret key, so it does not prevent an imposter from creating a message that claims to be from someone else and computing an MD5 checksum for that message. However, there are two ways to use MD5 in combination with a public key algorithm like RSA to implement message integrity. Both approaches overcome the performance problems inherent in using RSA alone.

The first method, which is commonly referred to as *keyed MD5*, works as follows. The sender generates a random key k and runs MD5 over the concatenation of the message (denoted m) and this key. In practice, the random key k is attached to either the front or back of the message for the purpose of running MD5; k is then removed from the message once MD5 is finished. The random key is then encrypted using RSA and the sender's private key. The original message, the MD5 checksum, and the encrypted version of the random key are then packaged together and sent to the receiver. The following summarizes the complete message transmitted by the sender:

$$m + \mathsf{MD5}(m + k) + \mathsf{E}(k, \mathit{private})$$

where $\mathsf{MD5}(s)$ represents applying the MD5 algorithm to string s, and $a+b$ denotes the concatenation of strings a and b.

The receiver recovers the random key using the purported sender's public RSA key and then applies MD5 to the concatenation of this random key and the body of the message. If the result matches the checksum sent with the message, then the message must have been sent by the participant who generated the random key.

MD5 with RSA Signature

The second method for using MD5 in combination with RSA works as follows. The sender runs MD5 over the original message it wants to protect, producing an MD5 checksum. It then signs this checksum with its own private RSA key. That is, the sender does not sign the entire message, it just signs the checksum. The original message, the MD5 checksum, and the RSA signature for the checksum are then transmitted. Using the same notation as above, this means that the sender transmits

$$m + \mathsf{MD5}(m) + \mathsf{E}(\mathsf{MD5}(m), \mathit{private}).$$

The receiver verifies the message by making sure that the signature is correct, that is, by decrypting the signature with the sender's public key and comparing the result with the MD5 checksum sent with the message. The two should be equal.

Using Digital Signatures for Key Distribution

We mentioned above that digital signatures can be used to build a mesh of trust by which keys can be distributed, as an alternative to a strict hierarchy. Now that we know how digital signatures work, providing us with a way to say "this message was generated by participant X and has not been altered," we can describe the mesh approach.

To begin, suppose participants A and B exchange public keys while they are in the same room; that is, they can verify that the keys really belong to the right people. At some later time, A exchanges public keys with C. A can now use his or her private key to sign the public key of C and send it to B. In other words, A sends a message that says, in effect, "I was in a room with C and I believe that this is C's public key," and signs this message using his or her own private key. Because B has a trustworthy copy of A's public key, B can verify the signature on this message. As long as B trusts that A was not fooled into mistakenly signing C's key, B now has a trustworthy copy of C's public key.

Thus, a reasonable strategy is for C to get his or her key signed by lots of people, and with luck, anyone with whom C wants to communicate will be able to find one person they trust in the set of people who have signed C's public key. Note that there can be arbitrarily many links in the chain of trust, but if one person in the chain signs a key when they are not sure it really belongs to the right person, then the chain of trust is broken. As an aside, note that key-signing parties—a lot of As, Bs, and Cs in a room with their laptops—have become a regular feature of IETF meetings. This approach to key management is used by the Pretty Good Privacy (PGP) suite of software.

Security Support in IPv6

IPv6 includes hooks to support authentication, message integrity, and encryption. First, IPv6 provides an Authentication extension header, which contains only two fields of interest: a Security Parameters Index (SPI) and some amount of Authentication Data. This provides a very general way to support a wide range of authentication algorithms and key management strategies without needing to redefine the header format. It is assumed that any two parties (hosts or routers) that wish to authenticate packets from each other will be able to agree on everything needed to provide authentication, notably an algorithm and a key. Once they have done this, the sending host uses the SPI to identify the appropriate set of agreements to the receiver. The sender then uses the agreed-upon algorithm and key to calculate a checksum over the entire datagram, and the receiver uses the

7.4 Summary

This chapter has described three different data manipulations that applications apply to their data before sending it over a network. Unlike the protocols described earlier in this book, which you can think of as processing *messages*, these three transformations process *data*.

The first data manipulation is presentation formatting, where the problem is encoding the different types of data that application programs compute on—e.g., integers, floating-point numbers, character strings, arrays, structures—into packets that can be transmitted over a network. This involves both translating between machine and network byte order, and linearizing compound data structures. We outlined the design space for presentation formatting and discussed three specific mechanisms that fall on different points in this design space: XDR, ASN.1, and NDR.

The second data manipulation is compression, which is concerned with reducing the bandwidth required to transmit different types of data. Compression algorithms can be either lossless or lossy, with lossy algorithms being most appropriate for image and video data. JPEG and MPEG are examples of lossy compression protocols for still images and video data, respectively.

The third data manipulation is encryption, which is used to protect the privacy of data sent over a network. We looked at two basic encryption algorithms: shared key algorithms (e.g., DES) and public key algorithms (e.g., RSA). As we have seen, encryption algorithms actually play a much bigger role in network security than just supporting privacy; they also form the basis for protocols that are used to ensure authentication and message integrity.

same algorithm and key to verify the checksum. If the packet has been modified in any way, the checksum will fail, which is how we achieve integrity. If someone other than the purported sender tries to send the packet, since they do not know the key to the algorithm, they will be unable to forge the correct checksum.

A similarly general approach is used for encryption. Specifically, another extension header, called the Encapsulating Security Payload (ESP) header, is used. All headers that precede the ESP header, most notably the IPv6 header, are unencrypted, so that routers do not need to decrypt packets before forwarding them. The only mandatory field of the ESP header is the SPI, which again provides an identifier by which the receiver can look up the algorithm and key needed to decrypt the data. Everything that follows the ESP header in the packet is then encrypted.

OPEN ISSUE

Presentation—Layer of the 90s

At one time, data manipulations such as the ones discussed in this chapter were treated as an afterthought by networking people. There were various reasons for this. In the case of presentation formatting, the network was so slow that no one cared how bad the encoding strategy was. In the case of compression, it is the recent advances in CD-ROM and graphical I/O devices (i.e., the growing popularity of multimedia) that have made the compression of images a fruitful area to pursue. In the case of security, it is the recent explosion of the World Wide Web, and the implications of using the Web for commerce, that have made people more security conscious. (Of course, the military has always been highly interested in security.) Today, however, one could argue that the data manipulation level of the network architecture is where the action is. In short, it is the set of data manipulation functions available to application programs that is limiting what we can do with computer networks.

Network security is a good example of services that are just around the corner. Using the basic cryptography algorithms like DES, RSA, and MD5, people are designing security protocols that do much more than ensure privacy, authentication, and message integrity. For example, there are now protocols that support anonymous digital cash, certified mail, secure elections, and simultaneous contract signing.

In the case of data compression, formats like MPEG were not specifically designed for use on a network; they were designed to provide a format for storing video on disk. It is not completely clear that MPEG makes sense for sending video over a network. For example, if a B or P frame is dropped by the network, then it is possible to simply replay the previous frame without seriously compromising the video; 1 frame out of 30 is no big deal. On the other hand, a lost I frame has serious consequences—none of the subsequent B and P frames can be processed without it. Thus, losing an I frame would result in losing multiple frames of the video. While you could retransmit the missing I frame, the resulting delay would probably not be acceptable in a real-time video conference. One interesting avenue of research is to have the application "teach" the network how to treat different types of messages; for example, it is OK to drop messages carrying B and P frames but not messages carrying I frames. Note, however, that this is a significant departure from the current model in which all messages are treated equally by the network.

One final issue to consider is the relationship among these three transformations. In most cases, you would perform presentation encoding before compression or encryption—the application first packs and linearizes a data structure before trying to compress or encrypt it. Between compression and encryption, you would always compress before encrypting. This is for two reasons. First, if the encryption algorithm is a good one, then there is no obvious redundancy for the compression algorithm to remove. The second reason is that by first compressing the data, you are able to make the ciphertext that results from encryption even more obscure, and therefore more difficult to break.

Ex.1022 / Page 59 of 63
Intel Corporation and Lenovo (United States) Inc.

FURTHER READING

Our recommended reading list for this chapter includes two papers on compression and two papers on network security. The two articles on compression give an overview of the JPEG and MPEG standards, respectively. Their main value is in explaining the various factors that shaped the standards. The two security-related papers, taken together, give a good overview of the topic. The article by Lampson et al. contains a formal treatment of security, while the Satyanarayanan paper gives a nice description of how a secure system is designed in practice.

The final paper on the reading list discusses the importance of data manipulations to network performance and gives two architectural principles that should guide the design of future protocols.

- Wallace, G. K. The JPEG still picture compression standard. *Communications of the ACM* 34(1):30–44, April 1991.
- Le Gall, D. MPEG: A video compression standard for multimedia applications. *Communications of the ACM* 34(1):46–58, April 1991.
- Lampson, B., et al. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems* 10(4):265–310, November 1992.
- Satyanarayanan, M. Integrating security in a large distributed system. *ACM Transactions on Computer Systems* 7(3):247–280, August 1989.
- Clark, D., and D. Tennenhouse. Architectural considerations for a new generation of protocols. *Proceedings of the SIGCOMM '91 Symposium*, pages 200–208, September 1991.

Unfortunately, there is no single paper that gives a comprehensive treatment of presentation formatting. Aside from the XDR, ASN.1/BER, and NDR specifications (see Srinivasan [Sri95b], the CCIT recommendations [dTeT92a, dTeT92b], and the Open Software Foundation [Ope94]), three other papers cover topics related to presentation formatting: those by O'Malley et al. [OPM94], Lin [Lin93], and Chen et al. [CLNZ89]. All three discuss performance-related issues.

On the topic of compression, a good place to start is with Huffman encoding, which was originally defined in [Huf52]. The original LZ algorithm is presented in Ziv and Lempel [ZL77], and an improved version of that algorithm by the same authors can be found in [ZL78]. Both of these papers are of a theoretical nature. The work that brought the LZ approach into widespread practice can be found in Welch [Wel84]. For a more complete overview of the topic of compression, Nelson's article [Nel92] is recommended. You can also learn about compression in any of several recent books on multimedia. We recommend Witten et al. [WMB94], which has an extremely high science-to-hype ratio, and Buford [Buf94], which is a collection of contributed chapters that span the range of multimedia topics.

On the topic of network security, there are several good books to choose from. We recommend Schneier [Sch94] and Kaufman et al. [KPS95]. The former gives a comprehensive

treatment of the topic, including sample code, while the latter gives a very readable overview of the subject.

Finally, we recommend the following two live references:

- http://roger-rabbit.cs.berkeley.edu/mpeg/index.html: a collection of MPEG-related programs, some of which are used in the following exercises;
- ftp://cert.org/pub: a collection of security-related notices posted by the Computer Emergency Response Team (CERT).

EXERCISES

1 Consider the following C structure definitions:

```
#define  MAXSTR 100

struct date {
    char  month[MAXSTR];
    int   day;
    int   year;
};

struct employee {
    char   name[MAXSTR];
    int    ssn;
    struct date *hireday;
    int    salary_history[10];
    int    num_raises;
}
```

where num_raises + 1 corresponds to the number of valid entries in array salary_history. Show the on-the-wire representation of structure employee that is generated by XDR.

2 For the data structures given in the previous problem, give the XDR routine that encodes/decodes these structures. If you have XDR available to you, run this routine and measure how long it takes to encode and decode an example instance of structure employee.

3 Using Unix utilities like bcopy and htonl, implement a routine that generates the same on-the-wire representation of the structures given in the problem 1 as XDR does. Compare the performance of your "by hand" encoder/decoder with the corresponding XDR routines.

4 Retrieve the Universal Stub Compiler (USC) byte-swapping program from ftp.cs.arizona.edu, and write the USC specification that encodes/decodes the data struc-

tures given in problem 1. (USC does not handle pointers, so you will need to encode/decode that part by hand.) Compare the performance of your USC encoder/decoder with the XDR and by-hand versions from the previous two exercises.

5 Use USC, XDR, and htonl to encode a 1000-element array of integers. Measure and compare the performance of each. How do the three compare to a simple loop that reads and writes a 1000-element array of integers? Perform the experiment on a computer for which the native byte order is the same as the network byte order, as well as on a computer for which the native byte order and the network byte order are different.

6 Give the ASN.1 encoding for the following three integers:

(a) 101

(b) 10,120

(c) 21,645

7 Give the big-endian and little-endian representation for the integers from the previous problem.

8 Determine the byte order for the following architectures: Alpha, MIPS, Sparc, and PA-RISC.

9 XDR is used to encode/decode the header for the SunRPC protocol described in Chapter 6. What difficulties might arise in using XDR in this way?

10 Why might you want to compress your data before doing presentation formatting?

11 Write a routine that implements RLE and DPCM compression. Apply these routines to different types of data, and report the compression ratios that each algorithm achieves.

12 Experiment with the Unix compress utility. What compression ratios is this program able to achieve? See if you can generate data files for which compress can achieve 5:1 or 10:1 compression ratios.

13 Think about what functions might reasonably be expected from a video standard: fast forward, editing capabilities, random access, and so on. (See the paper by LeGall, "MPEG: A video compression standard for multimedia applications," given in this chapter's recommended reading list, for further ideas.) Explain MPEG's design in terms of these features.

14 The frequency of I frames in the MPEG stream defines the granularity of fast forward and reverse. Explain why.

15 Use mpeg_play to play an MPEG-encoded video. Experiment with options, particularly -nob and -nop, which are used to omit the B and P frames, respectively, from the stream. What are the visible effects of omitting these frames?

16 The mpeg_stat program can be used to display statistics for video streams. Use it to determine, for several streams:

(a) number and sequence of I, B, and P frames.

(b) average compression rate for the entire video.

(c) average compression rate for each type of frame.

17 Write a program that implements forward and backward DCT. Run the program on a sample grayscale image. (It would be easiest to use PBM to read and write images.) Since DCT is lossless, the image output by the program should match the input. Now modify your program so that it throws away or changes some of the higher-frequency components and see how the output image is affected.

18 The Unix des command encrypts and decrypts data using the Data Encryption Standard algorithm. If it is available on your system, read the man page and experiment with it. Measure how fast it is able to encrypt or decrypt data.

19 Prove that the RSA decryption algorithm recovers the original message.

20 In the three-way handshake described as an authentication mechanism, why is the server unsure of the client's identity until it receives the third message?

21 Figure 7.21 shows CBC encryption. Give the corresponding diagram for decryption.

22 Show how two-way authentication works using RSA.

23 Learn about the Clipper encryption scheme. What are the pros and cons of Clipper?

24 What sort of attacks are authentication schemes like Kerberos susceptible to?

Ex.1022 / Page 63 of 63
Intel Corporation and Lenovo (United States) Inc.