TCP/IP Illustrated, Volume 1

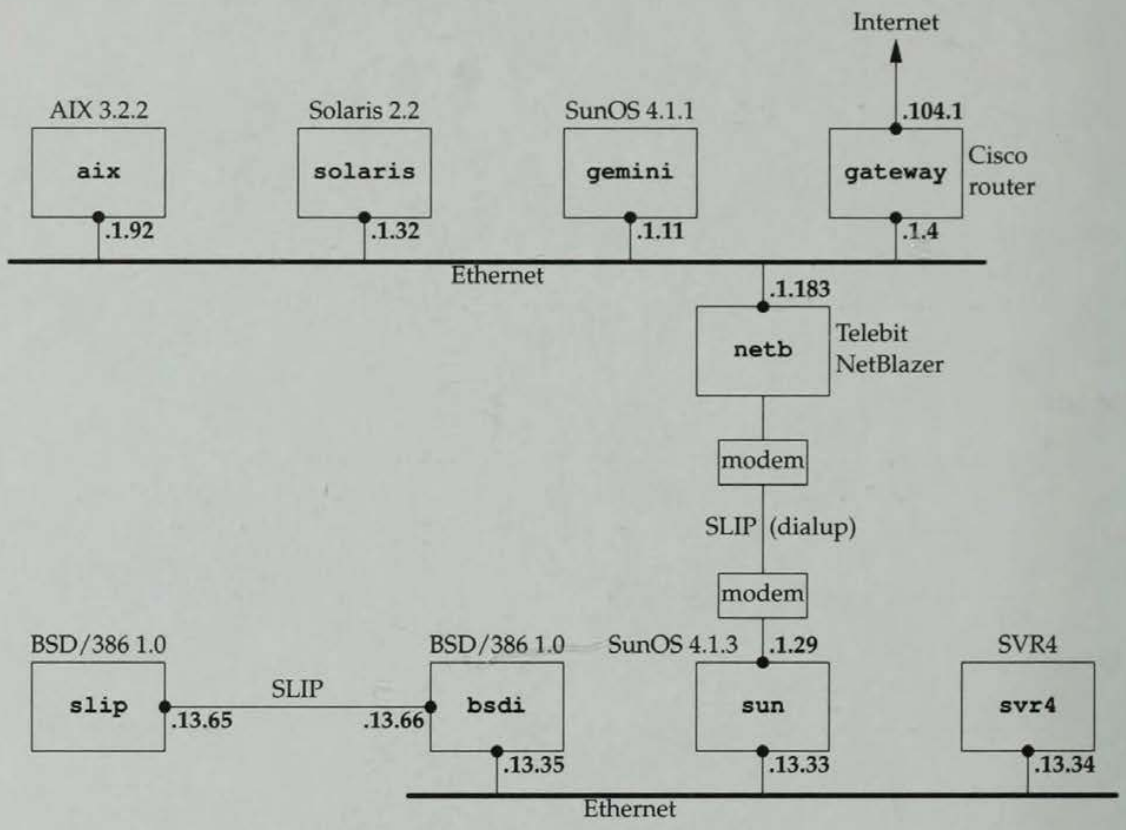
The Protocols

W. Richard Stevens



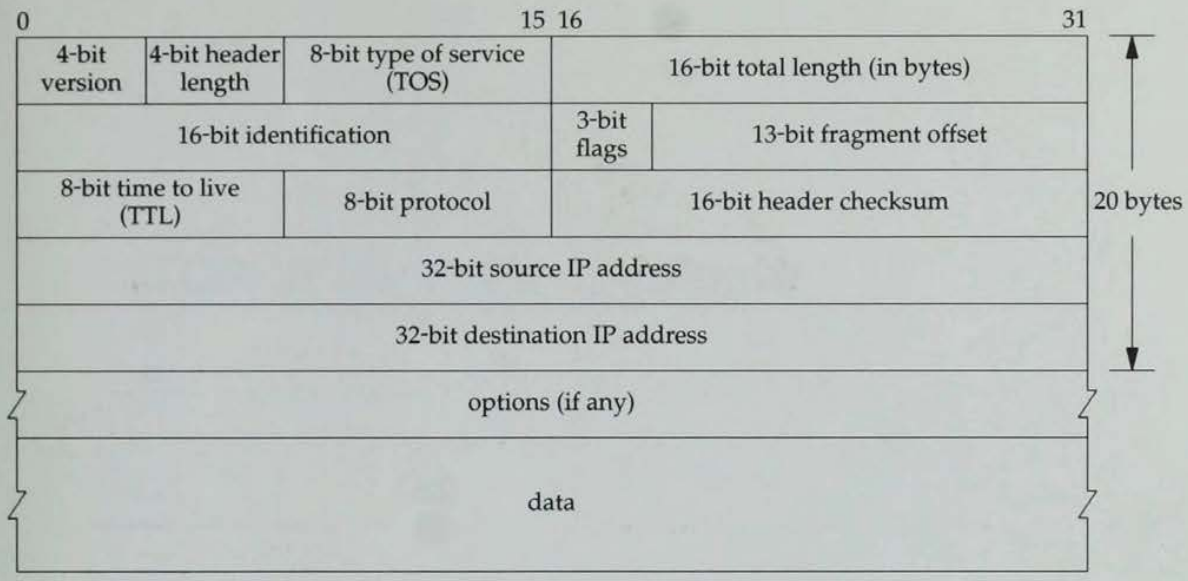
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

SAMSUNG 1013

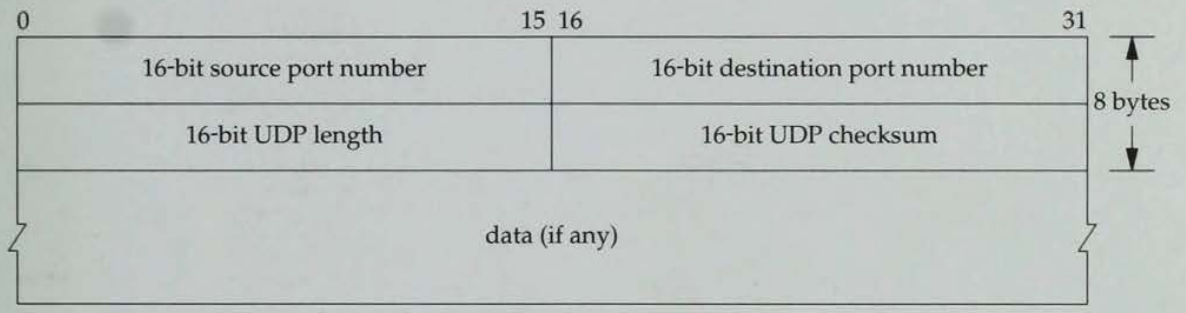


Portion of the class B network **140.252** used for all the examples in the text.
All the hosts are in the `tuc.noao.edu` domain.

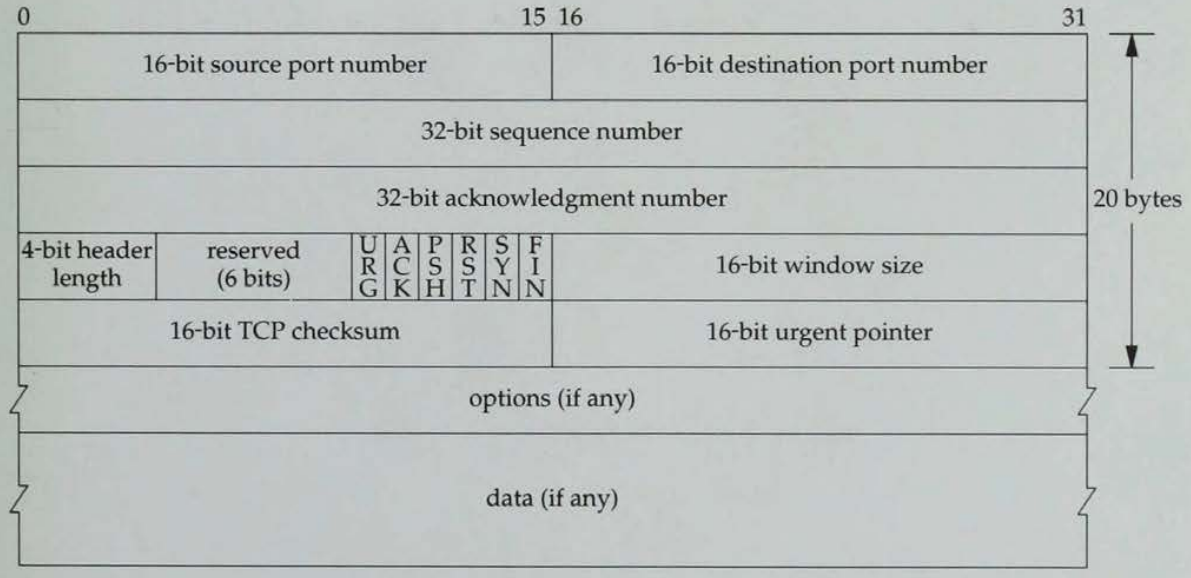
IP Header

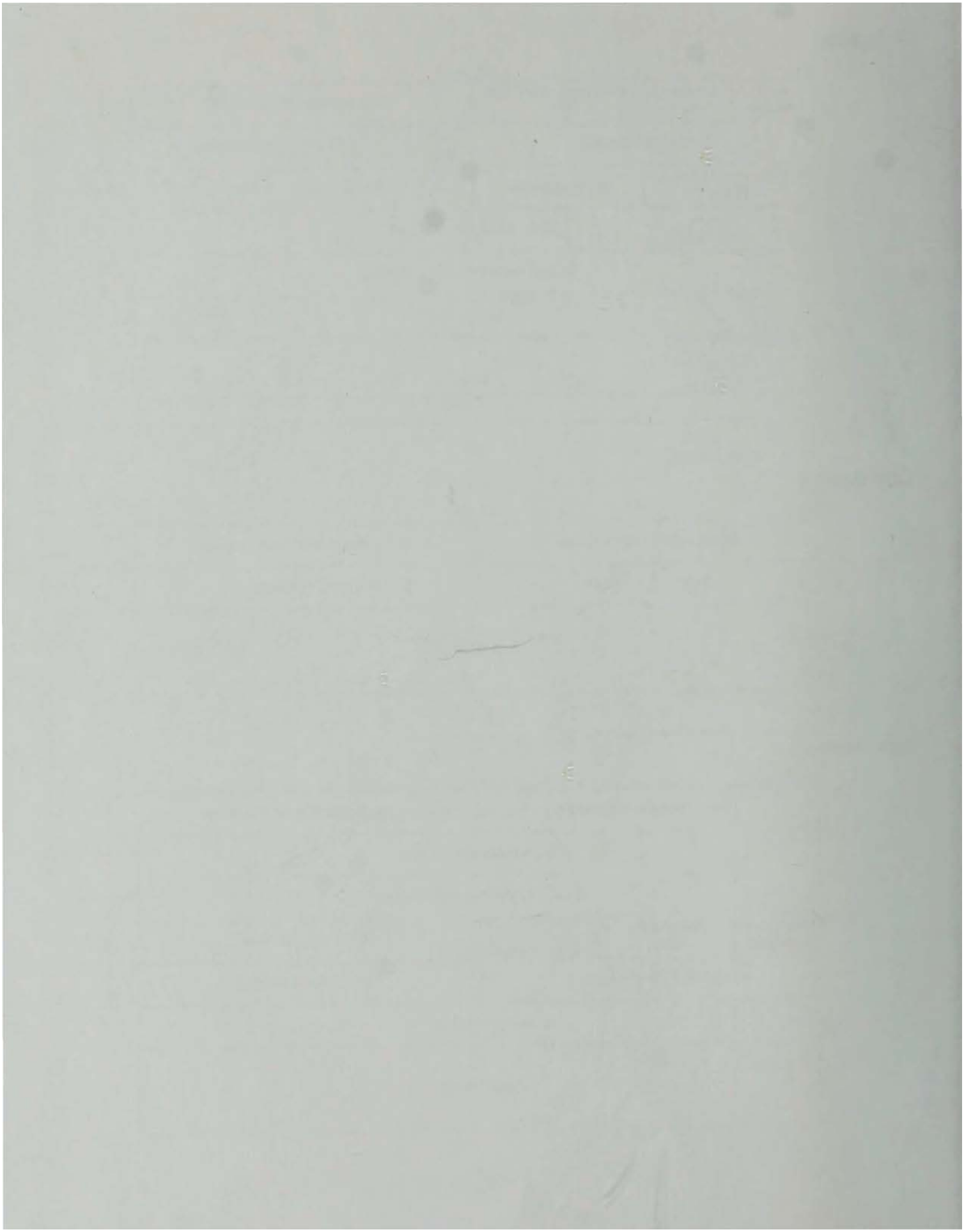


UDP Header



TCP Header





TCP/IP Illustrated, Volume 1

Addison-Wesley Professional Computing Series

Brian W. Kernighan, Consulting Editor

Ken Arnold/John Peyton, *A C User's Guide to ANSI C*
Tom Cargill, *C++ Programming Style*
David A. Curry, *UNIX® System Security: A Guide for Users and System Administrators*
William R. Cheswick/Steven M. Bellovin, *Firewalls and Internet Security: Repelling the Wily Hacker*
Scott Meyers, *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*
Robert B. Murray, *C++ Strategies and Tactics*
John K. Ousterhout, *Tcl and the Tk Toolkit*
Craig Partridge, *Gigabit Networking*
Radia Perlman, *Interconnections: Bridges and Routers*
David M. Piscitello/A. Lyman Chapin, *Open Systems Networking: TCP/IP and OSI*
Stephen A. Rago, *UNIX® System V Network Programming*
Curt Schimmel, *UNIX® Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*
W. Richard Stevens, *Advanced Programming in the UNIX® Environment*
W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*

TCP/IP Illustrated, Volume 1

The Protocols

W. Richard Stevens



ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

UNIX is a technology trademark of X/Open Company, Ltd.

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

Corporate & Professional Publishing Group
Addison-Wesley Publishing Company
One Jacob Way
Reading, Massachusetts 01867

Library of Congress Cataloging-in-Publication Data
Stevens, W. Richard
TCP/IP Illustrated: the protocols/W. Richard Stevens.
p. cm. — (Addison-Wesley professional computing series)
Includes bibliographical references and index.
ISBN 0-201-63346-9 (v. 1)
1.TCP/IP (Computer network protocol) I. Title. II. Series.
TK5105.55S74 1994
004.6'2—dc20

Copyright © 1994 by Addison-Wesley Publishing Company

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

ISBN 0-201-63346-9
Text printed on recycled paper.

3 4 5 6 7 8 9 CRW 97969594
Third printing May 1994

To Brian Kernighan and John Wait,
for their encouragement, faith, and support
over the past 5 years.

Praise for *TCP/IP Illustrated, Volume 1: The Protocols*

"This is sure to be the bible for TCP/IP developers and users. Within minutes of picking up the text, I encountered several scenarios which had tripped-up both my colleagues and myself in the past. Stevens reveals many of the mysteries once held tightly by the ever-elusive networking gurus. Having been involved in the implementation of TCP/IP for some years now, I consider this by far the finest text to date."

— Robert A. Ciampa, Network Engineer, Synernetics, division of 3COM

"While all of Stevens' books are readable and technically excellent, this new opus is awesome. Although many books describe the TCP/IP protocols, Stevens provides a level of depth and real-world detail lacking from the competition. He puts the reader inside TCP/IP using a visual approach and shows the protocols in action."

— Steven Baker, Networking Columnist, *Unix Review*

"*TCP/IP Illustrated, Volume 1* is an excellent reference for developers, network administrators, or anyone who needs to understand TCP/IP technology. *TCP/IP Illustrated* is comprehensive in its coverage of TCP/IP topics, providing enough details to satisfy the experts while giving enough background and commentary for the novice."

— Bob Williams, V.P. Marketing, NetManage, Inc.

"... the difference is that Stevens wants to show as well as tell about the protocols. His principal teaching tools are straight-forward explanations, exercises at the ends of chapters, byte-by-byte diagrams of headers and the like, and listings of actual traffic as examples."

— Walter Zintz, *UnixWorld*

"Much better than theory only ... W. Richard Stevens takes a multihost-based configuration and uses it as a travelogue of TCP/IP examples with illustrations. *TCP/IP Illustrated, Volume 1* is based on practical examples that reinforce the theory — distinguishing this book from others on the subject, and making it both readable and informative."

— Peter M. Haverlock, Consultant, IBM TCP/IP Development

"The diagrams he uses are excellent and his writing style is clear and readable. In sum, Stevens has made a complex topic easy to understand. This book merits everyone's attention. Please read it and keep it on your bookshelf."

— Elizabeth Zinkann, *Sys Admin*

"W. Richard Stevens has produced a fine text and reference work. It is well organized and very clearly written with, as the title suggests, many excellent illustrations exposing the intimate details of the logic and operation of IP, TCP, and the supporting cast of protocols and applications."

— Scott Bradner, Consultant, Harvard University OIT/NSD

Contents

Preface xv

Chapter 1. Introduction 1

1.1 Introduction 1
1.2 Layering 1
1.3 TCP/IP Layering 6
1.4 Internet Addresses 7
1.5 The Domain Name System 9
1.6 Encapsulation 9
1.7 Demultiplexing 11
1.8 Client–Server Model 12
1.9 Port Numbers 12
1.10 Standardization Process 14
1.11 RFCs 14
1.12 Standard, Simple Services 15
1.13 The Internet 16
1.14 Implementations 16
1.15 Application Programming Interfaces 17
1.16 Test Network 18
1.17 Summary 19

Chapter 2. Link Layer **21**

2.1 Introduction 21
2.2 Ethernet and IEEE 802 Encapsulation 21
2.3 Trailer Encapsulation 23
2.4 SLIP: Serial Line IP 24
2.5 Compressed SLIP 25
2.6 PPP: Point-to-Point Protocol 26
2.7 Loopback Interface 28
2.8 MTU 29
2.9 Path MTU 30
2.10 Serial Line Throughput Calculations 30
2.11 Summary 31

Chapter 3. IP: Internet Protocol **33**

3.1 Introduction 33
3.2 IP Header 34
3.3 IP Routing 37
3.4 Subnet Addressing 42
3.5 Subnet Mask 43
3.6 Special Case IP Addresses 45
3.7 A Subnet Example 46
3.8 `ifconfig` Command 47
3.9 `netstat` Command 49
3.10 IP Futures 49
3.11 Summary 50

Chapter 4. ARP: Address Resolution Protocol **53**

4.1 Introduction 53
4.2 An Example 54
4.3 ARP Cache 56
4.4 ARP Packet Format 56
4.5 ARP Examples 57
4.6 Proxy ARP 60
4.7 Gratuitous ARP 62
4.8 `arp` Command 63
4.9 Summary 63

Chapter 5. RARP: Reverse Address Resolution Protocol **65**

5.1 Introduction 65
5.2 RARP Packet Format 65
5.3 RARP Examples 66
5.4 RARP Server Design 67
5.5 Summary 68

Chapter 6. ICMP: Internet Control Message Protocol 69

6.1 Introduction 69
6.2 ICMP Message Types 70
6.3 ICMP Address Mask Request and Reply 72
6.4 ICMP Timestamp Request and Reply 74
6.5 ICMP Port Unreachable Error 77
6.6 4.4BSD Processing of ICMP Messages 81
6.7 Summary 83

Chapter 7. Ping Program 85

7.1 Introduction 85
7.2 Ping Program 85
7.3 IP Record Route Option 91
7.4 IP Timestamp Option 95
7.5 Summary 96

Chapter 8. Traceroute Program 97

8.1 Introduction 97
8.2 Traceroute Program Operation 97
8.3 LAN Output 99
8.4 WAN Output 102
8.5 IP Source Routing Option 104
8.6 Summary 109

Chapter 9. IP Routing 111

9.1 Introduction 111
9.2 Routing Principles 112
9.3 ICMP Host and Network Unreachable Errors 117
9.4 To Forward or Not to Forward 119
9.5 ICMP Redirect Errors 119
9.6 ICMP Router Discovery Messages 123
9.7 Summary 125

Chapter 10. Dynamic Routing Protocols 127

10.1 Introduction 127
10.2 Dynamic Routing 127
10.3 Unix Routing Daemons 128
10.4 RIP: Routing Information Protocol 129
10.5 RIP Version 2 136
10.6 OSPF: Open Shortest Path First 137
10.7 BGP: Border Gateway Protocol 138
10.8 CIDR: Classless Interdomain Routing 140
10.9 Summary 141

Chapter 11. UDP: User Datagram Protocol 143

11.1 Introduction 143
11.2 UDP Header 144
11.3 UDP Checksum 144
11.4 A Simple Example 147
11.5 IP Fragmentation 148
11.6 ICMP Unreachable Error (Fragmentation Required) 151
11.7 Determining the Path MTU Using Traceroute 153
11.8 Path MTU Discovery with UDP 155
11.9 Interaction Between UDP and ARP 157
11.10 Maximum UDP Datagram Size 159
11.11 ICMP Source Quench Error 160
11.12 UDP Server Design 162
11.13 Summary 167

Chapter 12. Broadcasting and Multicasting 169

12.1 Introduction 169
12.2 Broadcasting 171
12.3 Broadcasting Examples 172
12.4 Multicasting 175
12.5 Summary 178

Chapter 13. IGMP: Internet Group Management Protocol 179

13.1 Introduction 179
13.2 IGMP Message 180
13.3 IGMP Protocol 180
13.4 An Example 183
13.5 Summary 186

Chapter 14. DNS: The Domain Name System 187

14.1 Introduction 187
14.2 DNS Basics 188
14.3 DNS Message Format 191
14.4 A Simple Example 194
14.5 Pointer Queries 198
14.6 Resource Records 201
14.7 Caching 203
14.8 UDP or TCP 206
14.9 Another Example 206
14.10 Summary 208

Chapter 15. TFTP: Trivial File Transfer Protocol 209

15.1 Introduction 209
15.2 Protocol 209
15.3 An Example 211
15.4 Security 213
15.5 Summary 213

Chapter 16. BOOTP: Bootstrap Protocol 215

16.1 Introduction 215
16.2 BOOTP Packet Format 215
16.3 An Example 218
16.4 BOOTP Server Design 219
16.5 BOOTP Through a Router 220
16.6 Vendor-Specific Information 221
16.7 Summary 222

Chapter 17. TCP: Transmission Control Protocol 223

17.1 Introduction 223
17.2 TCP Services 223
17.3 TCP Header 225
17.4 Summary 227

Chapter 18. TCP Connection Establishment and Termination 229

18.1 Introduction 229
18.2 Connection Establishment and Termination 229
18.3 Timeout of Connection Establishment 235
18.4 Maximum Segment Size 236
18.5 TCP Half-Close 238
18.6 TCP State Transition Diagram 240
18.7 Reset Segments 246
18.8 Simultaneous Open 250
18.9 Simultaneous Close 252
18.10 TCP Options 253
18.11 TCP Server Design 254
18.12 Summary 260

Chapter 19. TCP Interactive Data Flow 263

19.1 Introduction 263
19.2 Interactive Input 263
19.3 Delayed Acknowledgments 265
19.4 Nagle Algorithm 267
19.5 Window Size Advertisements 274
19.6 Summary 274

Chapter 20. TCP Bulk Data Flow 275

20.1 Introduction 275
20.2 Normal Data Flow 275
20.3 Sliding Windows 280
20.4 Window Size 282
20.5 PUSH Flag 284
20.6 Slow Start 285
20.7 Bulk Data Throughput 286
20.8 Urgent Mode 292
20.9 Summary 296

Chapter 21. TCP Timeout and Retransmission 297

21.1 Introduction 297
21.2 Simple Timeout and Retransmission Example 298
21.3 Round-Trip Time Measurement 299
21.4 An RTT Example 301
21.5 Congestion Example 306
21.6 Congestion Avoidance Algorithm 310
21.7 Fast Retransmit and Fast Recovery Algorithm 312
21.8 Congestion Example (Continued) 313
21.9 Per-Route Metrics 316
21.10 ICMP Errors 317
21.11 Repacketization 320
21.12 Summary 321

Chapter 22. TCP Persist Timer 323

22.1 Introduction 323
22.2 An Example 323
22.3 Silly Window Syndrome 325
22.4 Summary 330

Chapter 23. TCP Keepalive Timer 331

23.1 Introduction 331
23.2 Description 332
23.3 Keepalive Examples 333
23.4 Summary 337

Chapter 24. TCP Futures and Performance 339

24.1 Introduction 339
24.2 Path MTU Discovery 340
24.3 Long Fat Pipes 344
24.4 Window Scale Option 347

24.5 Timestamp Option 349
24.6 PAWS: Protection Against Wrapped Sequence Numbers 351
24.7 T/TCP: A TCP Extension for Transactions 351
24.8 TCP Performance 354
24.9 Summary 356

Chapter 25. SNMP: Simple Network Management Protocol 359

25.1 Introduction 359
25.2 Protocol 360
25.3 Structure of Management Information 363
25.4 Object Identifiers 364
25.5 Introduction to the Management Information Base 365
25.6 Instance Identification 367
25.7 Simple Examples 370
25.8 Management Information Base (Continued) 372
25.9 Additional Examples 382
25.10 Traps 385
25.11 ASN.1 and BER 386
25.12 SNMP Version 2 387
25.13 Summary 388

Chapter 26. Telnet and Rlogin: Remote Login 389

26.1 Introduction 389
26.2 Rlogin Protocol 391
26.3 Rlogin Examples 396
26.4 Telnet Protocol 401
26.5 Telnet Examples 406
26.6 Summary 417

Chapter 27. FTP: File Transfer Protocol 419

27.1 Introduction 419
27.2 FTP Protocol 419
27.3 FTP Examples 426
27.4 Summary 439

Chapter 28. SMTP: Simple Mail Transfer Protocol 441

28.1 Introduction 441
28.2 SMTP Protocol 442
28.3 SMTP Examples 448
28.4 SMTP Futures 452
28.5 Summary 459

Chapter 29. NFS: Network File System 461

29.1 Introduction 461
29.2 Sun Remote Procedure Call 461
29.3 XDR: External Data Representation 465
29.4 Port Mapper 465
29.5 NFS Protocol 467
29.6 NFS Examples 474
29.7 NFS Version 3 479
29.8 Summary 480

Chapter 30. Other TCP/IP Applications 481

30.1 Introduction 481
30.2 Finger Protocol 481
30.3 Whois Protocol 483
30.4 Archie, WAIS, Gopher, Veronica, and WWW 484
30.5 X Window System 486
30.6 Summary 490

Appendix A. The `tcpdump` Program 491

A.1 BSD Packet Filter 491
A.2 SunOS Network Interface Tap 493
A.3 SVR4 Data Link Provider Interface 494
A.4 `tcpdump` Output 495
A.5 Security Considerations 496
A.6 Socket Debug Option 496

Appendix B. Computer Clocks 499

Appendix C. The `sock` Program 503

Appendix D. Solutions to Selected Exercises 507

Appendix E. Configurable Options 525

E.1 BSD/386 Version 1.0 526
E.2 SunOS 4.1.3 527
E.3 System V Release 4 529
E.4 Solaris 2.2 529
E.5 AIX 3.2.2 536
E.6 4.4BSD 537

Appendix F. Source Code Availability 539

Bibliography 543

Index 555

Preface

Introduction

This book describes the TCP/IP protocol suite, but from a different perspective than other texts on TCP/IP. Instead of just describing the protocols and what they do, we'll use a popular diagnostic tool to watch the protocols in action. Seeing how the protocols operate in varying circumstances provides a greater understanding of how they work and why certain design decisions were made. It also provides a look into the implementation of the protocols, without having to wade through thousands of lines of source code.

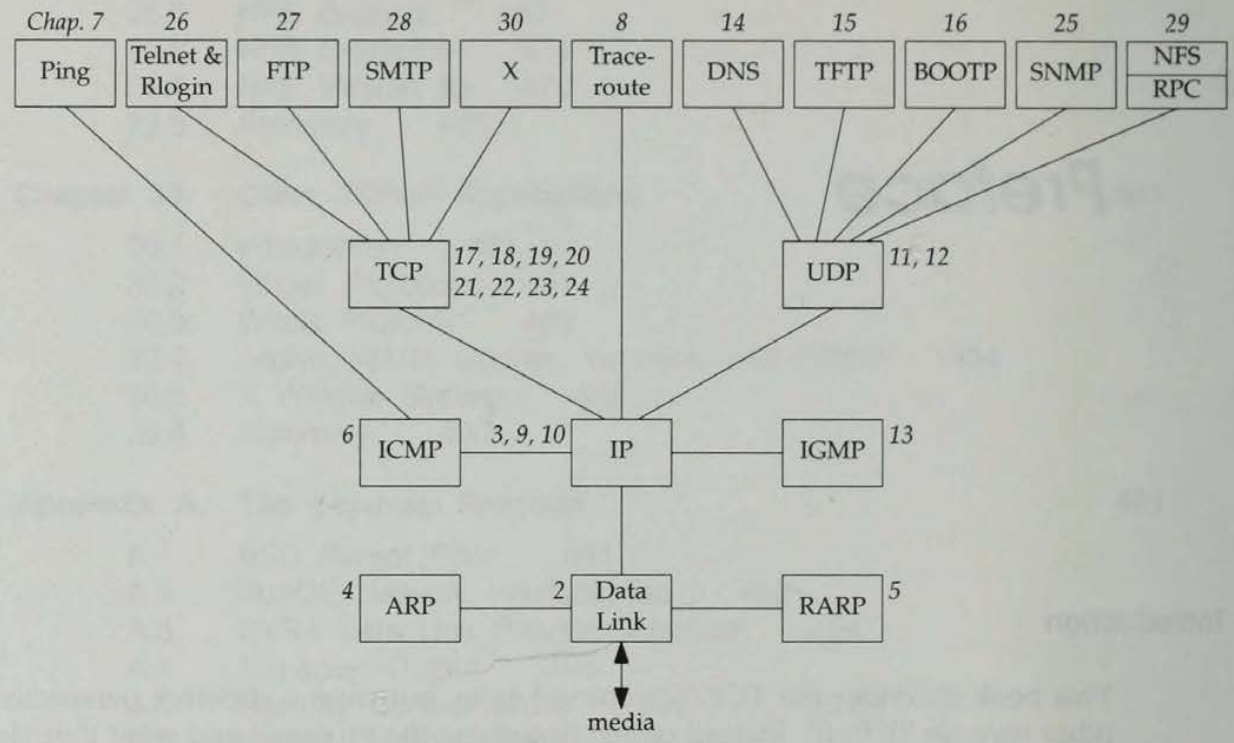
When networking protocols were being developed in the 1960s through the 1980s, expensive, dedicated hardware was required to see the packets going "across the wire." Extreme familiarity with the protocols was also required to comprehend the packets displayed by the hardware. Functionality of the hardware analyzers was limited to that built in by the hardware designers.

Today this has changed dramatically with the ability of the ubiquitous workstation to monitor a local area network [Mogul 1990]. Just attach a workstation to your network, run some publicly available software (described in Appendix A), and watch what goes by on the wire. While many people consider this a tool to be used for *diagnosing* network problems, it is also a powerful tool for *understanding* how the network protocols operate, which is the goal of this book.

This book is intended for anyone wishing to understand how the TCP/IP protocols operate: programmers writing network applications, system administrators responsible for maintaining computer systems and networks utilizing TCP/IP, and users who deal with TCP/IP applications on a daily basis.

Organization of the Book

The following figure shows the various protocols and applications that are covered. The italic number by each box indicates the chapter in which that protocol or application is described.



(Numerous fine points are missing from this figure that will be discussed in the appropriate chapter. For example, both the DNS and RPC use TCP, which we don't show.)

We take a bottom-up approach to the TCP/IP protocol suite. After providing a basic introduction to TCP/IP in Chapter 1, we will start at the link layer in Chapter 2 and work our way up the protocol stack. This provides the required background for later chapters for readers who aren't familiar with TCP/IP or networking in general.

This book also uses a functional approach instead of following a strict bottom-to-top order. For example, Chapter 3 describes the IP layer and the IP header. But there are numerous fields in the IP header that are best described in the context of an application that uses or is affected by a particular field. Fragmentation, for example, is best understood in terms of UDP (Chapter 11), the protocol often affected by it. The time-to-live field is fully described when we look at the Traceroute program in Chapter 8, because this field is the basis for the operation of the program. Similarly, many features of ICMP are described in the later chapters, in terms of how a particular ICMP message is used by a protocol or an application.

We also don't want to save all the good stuff until the end, so we describe TCP/IP applications as soon as we have the foundation to understand them. Ping and Traceroute are described after IP and ICMP have been discussed. The applications built on UDP (multicasting, the DNS, TFTP, and BOOTP) are described after UDP has been

examined. The TCP applications, however, along with network management, must be saved until the end, after we've thoroughly described TCP. This text focuses on how these applications use the TCP/IP protocols. We do not provide all the details on running these applications.

Readers

This book is self-contained and assumes no specific knowledge of networking or TCP/IP. Numerous references are provided for readers interested in additional details on specific topics.

This book can be used in many ways. It can be used as a self-study reference and covered from start to finish by someone interested in all the details on the TCP/IP protocol suite. Readers with some TCP/IP background might want to skip ahead and start with Chapter 7, and then focus on the specific chapters in which they're interested. Exercises are provided at the end of the chapters, and most solutions are in Appendix D. This is to maximize the usefulness of the text as a self-study reference.

When used as part of a one- or two-semester course in computer networking, the focus should be on IP (Chapters 3 and 9), UDP (Chapter 11), and TCP (Chapters 17–24), along with some of the application chapters.

Many forward and backward references are provided throughout the text, along with a thorough index, to allow individual chapters to be studied by themselves. A list of all the acronyms used throughout the text, along with the compound term for the acronym, appears on the inside back covers.

If you have access to a network you are encouraged to obtain the software used in this book (Appendix F) and experiment on your own. Hands-on experimentation with the protocols will provide the greatest knowledge (and make it more fun).

Systems Used for Testing

Every example in the book was run on an actual network and the resulting output saved in a file for inclusion in the text. Figure 1.11 (p. 18) shows a diagram of the different hosts, routers, and networks that are used. (This figure is also duplicated on the inside front cover for easy reference while reading the book.) This collection of networks is simple enough that the topology doesn't confuse the examples, and with four systems acting as routers, we can see the error messages generated by routers.

Most of the systems have a name that indicates the type of software being used: `bsdi`, `svr4`, `sun`, `solaris`, `aix`, `slip`, and so on. In this way we can identify the type of software that we're dealing with by looking at the system name in the printed output.

A wide range of different operating systems and TCP/IP implementations are used:

- BSD/386 Version 1.0 from Berkeley Software Design, Inc., on the hosts named `bsdi` and `slip`. This system is derived from the BSD Networking Software, Release 2.0. (We show the lineage of the various BSD releases in Figure 1.10 on p. 17.)

- Unix System V/386 Release 4.0 Version 2.0 from U.H. Corporation, on the host named `svr4`. This is vanilla SVR4 and contains the standard implementation of TCP/IP from Lachman Associates used with most versions of SVR4.
- SunOS 4.1.3 from Sun Microsystems, on the host named `sun`. The SunOS 4.1.x systems are probably the most widely used TCP/IP implementations. The TCP/IP code is derived from 4.2BSD and 4.3BSD.
- Solaris 2.2 from Sun Microsystems, on the host named `solaris`. The Solaris 2.x systems have a different implementation of TCP/IP from the earlier SunOS 4.1.x systems, and from SVR4. (This operating system is really SunOS 5.2, but is commonly called Solaris 2.2.)
- AIX 3.2.2 from IBM on the host named `aix`. The TCP/IP implementation is based on the 4.3BSD Reno release.
- 4.4BSD from the Computer Systems Research Group at the University of California at Berkeley, on the host `vangogh.cs.berkeley.edu`. This system has the latest release of TCP/IP from Berkeley. (This system isn't shown in the figure on the inside front cover, but is reachable across the Internet.)

Although these are all Unix systems, TCP/IP is operating system independent, and is available on almost every popular non-Unix system. Most of this text also applies to these non-Unix implementations, although some programs (such as Traceroute) may not be provided on all systems.

Typographical Conventions

When we display interactive input and output we'll show our typed input in a **`bold font`**, and the computer output `like this`. *Comments are added in italics.*

```
bsdi % telnet svr4 discard          connect to the discard server
Trying 140.252.13.34...             this line and next output by Telnet client
Connected to svr4.
```

Also, we always include the name of the system as part of the shell prompt (`bsdi` in this example) to show on which host the command was run.

> Throughout the text we'll use indented, parenthetical notes such as this to describe historical points or implementation details.

We sometimes refer to the complete description of a command in the Unix manual as in `ifconfig`(8). This notation, the name of the command followed by a number in parentheses, is the normal way of referring to Unix commands. The number in parentheses is the section number in the Unix manual of the "manual page" for the command, where additional information can be located. Unfortunately not all Unix systems organize their manuals the same, with regard to the section numbers used for various groupings of commands. We'll use the BSD-style section numbers (which is the same for BSD-derived systems such as SunOS 4.1.3), but your manuals may be organized differently.

Acknowledgments

Although the author's name is the only one to appear on the cover, the combined effort of many people is required to produce a quality text book. First and foremost is the author's family, who put up with the long and weird hours that go into writing a book. Thank you once again, Sally, Bill, Ellen, and David.

The consulting editor, Brian Kernighan, is undoubtedly the best in the business. He was the first one to read various drafts of the manuscript and mark it up with his infinite supply of red pens. His attention to detail, his continual prodding for readable prose, and his thorough reviews of the manuscript are an immense resource to a writer.

Technical reviewers provide a different point of view and keep the author honest by catching technical mistakes. Their comments, suggestions, and (most importantly) criticisms add greatly to the final product. My thanks to Steve Bellovin, Jon Crowcroft, Pete Haverlock, and Doug Schmidt for comments on the entire manuscript. Equally valuable comments were provided on portions of the manuscript by Dave Borman, Tony DeSimone, Bob Gilligan, Jeff Gitlin, John Gulbenkian, Tom Herbert, Mukesh Kacker, Barry Margolin, Paul Mockapetris, Burr Nelson, Steve Rago, James Risner, Chris Walquist, Phil Winterbottom, and Gary Wright. A special thanks to Dave Borman for his thorough review of all the TCP chapters, and to Bob Gilligan who should be listed as a coauthor for Appendix E.

An author cannot work in isolation, so I would like to thank the following persons for lots of small favors, especially by answering my numerous e-mail questions: Joe Godsil, Jim Hogue, Mike Karels, Paul Lucchina, Craig Partridge, Thomas Skibo, and Jerry Toporek.

This book is the result of my being asked lots of questions on TCP/IP for which I could find no quick, immediate answer. It was then that I realized that the easiest way to obtain the answers was to run small tests, forcing certain conditions to occur, and just watch what happens. I thank Pete Haverlock for asking the probing questions and Van Jacobson for providing so much of the publicly available software that is used in this book to answer the questions.

A book on networking needs a real network to work with along with access to the Internet. My thanks to the National Optical Astronomy Observatories (NOAO), especially Sidney Wolff, Richard Wolff, and Steve Grandi, for providing access to their networks and hosts. A special thanks to Steve Grandi for answering lots of questions and providing accounts on various hosts. My thanks also to Keith Bostic and Kirk McKusick at the U.C. Berkeley CSRG for access to the latest 4.4BSD system.

Finally, it is the publisher that pulls everything together and does whatever is required to deliver the final product to the readers. This all revolves around the editor, and John Wait is simply the best there is. Working with John and the rest of the professionals at Addison-Wesley is a pleasure. Their professionalism and attention to detail show in the end result.

Camera-ready copy of the book was produced by the author, a Troff die-hard, using the Groff package written by James Clark. I welcome electronic mail from any readers with comments, suggestions, or bug fixes.

Tucson, Arizona
October 1993

W. Richard Stevens
`rstevens@noao.edu`

1

Introduction

1.1 Introduction

The TCP/IP protocol suite allows computers of all sizes, from many different computer vendors, running totally different operating systems, to communicate with each other. It is quite amazing because its use has far exceeded its original estimates. What started in the late 1960s as a government-financed research project into packet switching networks has, in the 1990s, turned into the most widely used form of networking between computers. It is truly an *open system* in that the definition of the protocol suite and many of its implementations are publicly available at little or no charge. It forms the basis for what is called the *worldwide Internet*, or the *Internet*, a wide area network (WAN) of more than one million computers that literally spans the globe.

This chapter provides an overview of the TCP/IP protocol suite, to establish an adequate background for the remaining chapters. For a historical perspective on the early development of TCP/IP see [Lynch 1993].

1.2 Layering

Networking *protocols* are normally developed in *layers*, with each layer responsible for a different facet of the communications. A *protocol suite*, such as TCP/IP, is the combination of different protocols at various layers. TCP/IP is normally considered to be a 4-layer system, as shown in Figure 1.1.

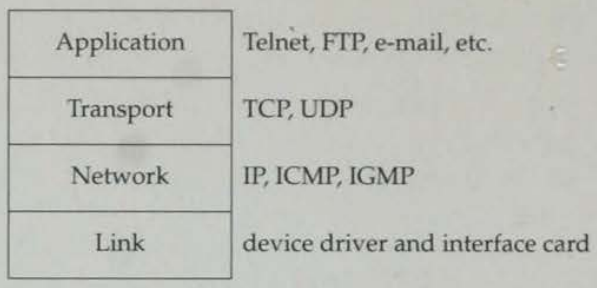


Figure 1.1 The four layers of the TCP/IP protocol suite.

Each layer has a different responsibility.

1. The *link* layer, sometimes called the *data-link* layer or *network interface* layer, normally includes the device driver in the operating system and the corresponding network interface card in the computer. Together they handle all the hardware details of physically interfacing with the cable (or whatever type of media is being used).

2. The *network* layer (sometimes called the *internet* layer) handles the movement of packets around the network. Routing of packets, for example, takes place here. IP (Internet Protocol), ICMP (Internet Control Message Protocol), and IGMP (Internet Group Management Protocol) provide the network layer in the TCP/IP protocol suite.

3. The *transport* layer provides a flow of data between two hosts, for the application layer above. In the TCP/IP protocol suite there are two vastly different transport protocols: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).

   TCP provides a reliable flow of data between two hosts. It is concerned with things such as dividing the data passed to it from the application into appropriately sized chunks for the network layer below, acknowledging received packets, setting timeouts to make certain the other end acknowledges packets that are sent, and so on. Because this reliable flow of data is provided by the transport layer, the application layer can ignore all these details.

   UDP, on the other hand, provides a much simpler service to the application layer. It just sends packets of data called *datagrams* from one host to the other, but there is no guarantee that the datagrams reach the other end. Any desired reliability must be added by the application layer.

   There is a use for each type of transport protocol, which we'll see when we look at the different applications that use TCP and UDP.

4. The *application* layer handles the details of the particular application. There are many common TCP/IP applications that almost every implementation provides:

   - Telnet for remote login,
   - FTP, the File Transfer Protocol,
   - SMTP, the Simple Mail Transfer protocol, for electronic mail,
   - SNMP, the Simple Network Management Protocol,

   and many more, some of which we cover in later chapters.

If we have two hosts on a local area network (LAN) such as an Ethernet, both running FTP, Figure 1.2 shows the protocols involved.

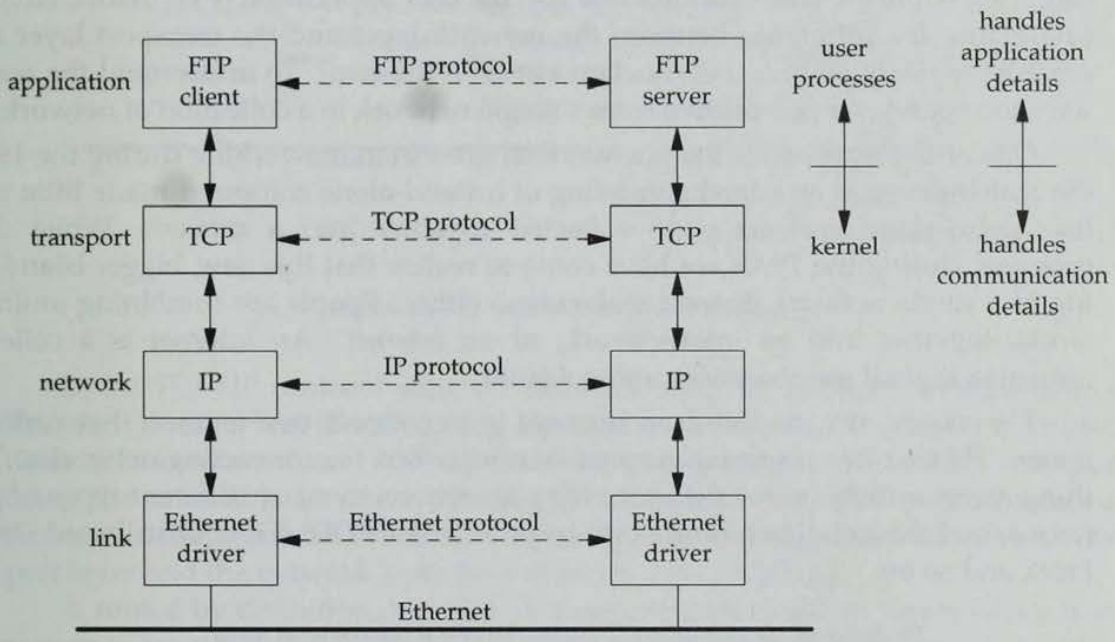


Figure 1.2 Two hosts on a LAN running FTP.

We have labeled one application box the FTP *client* and the other the FTP *server*. Most network applications are designed so that one end is the client and the other side the server. The server provides some type of service to clients, in this case access to files on the server host. In the remote login application, Telnet, the service provided to the client is the ability to login to the server's host.

Each layer has one or more protocols for communicating with its *peer* at the same layer. One protocol, for example, allows the two TCP layers to communicate, and another protocol lets the two IP layers communicate.

On the right side of Figure 1.2 we have noted that normally the application layer is a user process while the lower three layers are usually implemented in the kernel (the operating system). Although this isn't a requirement, it's typical and this is the way it's done under Unix.

There is another critical difference between the top layer in Figure 1.2 and the lower three layers. The application layer is concerned with the details of the application and not with the movement of data across the network. The lower three layers know nothing about the application but handle all the communication details.

We show four protocols in Figure 1.2, each at a different layer. FTP is an application layer protocol, TCP is a transport layer protocol, IP is a network layer protocol, and the Ethernet protocols operate at the link layer. The *TCP/IP protocol suite* is a combination of many protocols. Although the commonly used name for the entire protocol suite is TCP/IP, TCP and IP are only two of the protocols. (An alternative name is the *Internet Protocol Suite*.)

The purpose of the network interface layer and the application layer are obvious—the former handles the details of the communication media (Ethernet, token ring, etc.) while the latter handles one specific user application (FTP, Telnet, etc.). But on first glance the difference between the network layer and the transport layer is somewhat hazy. Why is there a distinction between the two? To understand the reason, we have to expand our perspective from a single network to a collection of networks.

One of the reasons for the phenomenal growth in networking during the 1980s was the realization that an island consisting of a stand-alone computer made little sense. A few stand-alone systems were collected together into a *network*. While this was progress, during the 1990s we have come to realize that this new, bigger island consisting of a single network doesn't make sense either. People are combining multiple networks together into an internetwork, or an *internet*. An internet is a collection of networks that all use the same protocol suite.

The easiest way to build an internet is to connect two or more networks with a *router*. This is often a special-purpose hardware box for connecting networks. The nice thing about routers is that they provide connections to many different types of physical networks: Ethernet, token ring, point-to-point links, FDDI (Fiber Distributed Data Interface), and so on.

> These boxes are also called *IP routers*, but we'll use the term *router*.
>
> Historically these boxes were called *gateways*, and this term is used throughout much of the TCP/IP literature. Today the term *gateway* is used for an application gateway: a process that connects two different protocol suites (say, TCP/IP and IBM's SNA) for one particular application (often electronic mail or file transfer).

Figure 1.3 shows an internet consisting of two networks: an Ethernet and a token ring, connected with a router. Although we show only two hosts communicating, with the router connecting the two networks, *any* host on the Ethernet can communicate with *any* host on the token ring.

In Figure 1.3 we can differentiate between an *end system* (the two hosts on either side) and an *intermediate system* (the router in the middle). The application layer and the transport layer use *end-to-end* protocols. In our picture these two layers are needed only on the end systems. The network layer, however, provides a *hop-by-hop* protocol and is used on the two end systems and every intermediate system.

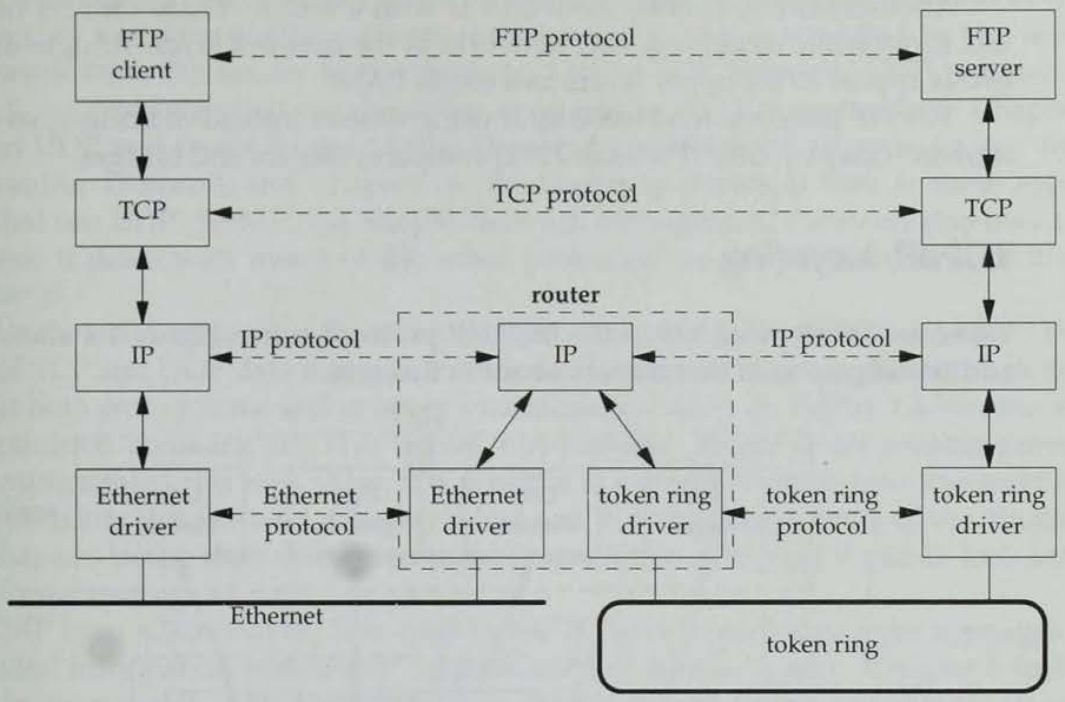


Figure 1.3 Two networks connected with a router.

In the TCP/IP protocol suite the network layer, IP, provides an unreliable service. That is, it does its best job of moving a packet from its source to its final destination, but there are no guarantees. TCP, on the other hand, provides a reliable transport layer using the unreliable service of IP. To provide this service, TCP performs timeout and retransmission, sends and receives end-to-end acknowledgments, and so on. The transport layer and the network layer have distinct responsibilities.

A router, by definition, has two or more network interface layers (since it connects two or more networks). Any system with multiple interfaces is called *multihomed*. A host can also be multihomed but unless it specifically forwards packets from one interface to another, it is not called a router. Also, routers need not be special hardware boxes that only move packets around an internet. Most TCP/IP implementations allow a multihomed host to act as a router also, but the host needs to be specifically configured for this to happen. In this case we can call the system either a host (when an application such as FTP or Telnet is being used) or a router (when it's forwarding packets from one network to another). We'll use whichever term makes sense given the context.

One of the goals of an internet is to hide all the details of the physical layout of the internet from the applications. Although this isn't obvious from our two-network internet in Figure 1.3, the application layers can't care (and don't care) that one host is on an Ethernet, the other on a token ring, with a router between. There could be 20 routers between, with additional types of physical interconnections, and the applications would run the same. This hiding of the details is what makes the concept of an internet so powerful and useful.

Another way to connect networks is with a *bridge*. These connect networks at the link layer, while routers connect networks at the network layer. Bridges makes multiple LANs appear to the upper layers as a single LAN.

TCP/IP internets tend to be built using routers instead of bridges, so we'll focus on routers. Chapter 12 of [Perlman 1992] compares routers and bridges.

1.3 TCP/IP Layering

There are more protocols in the TCP/IP protocol suite. Figure 1.4 shows some of the additional protocols that we talk about in this text.

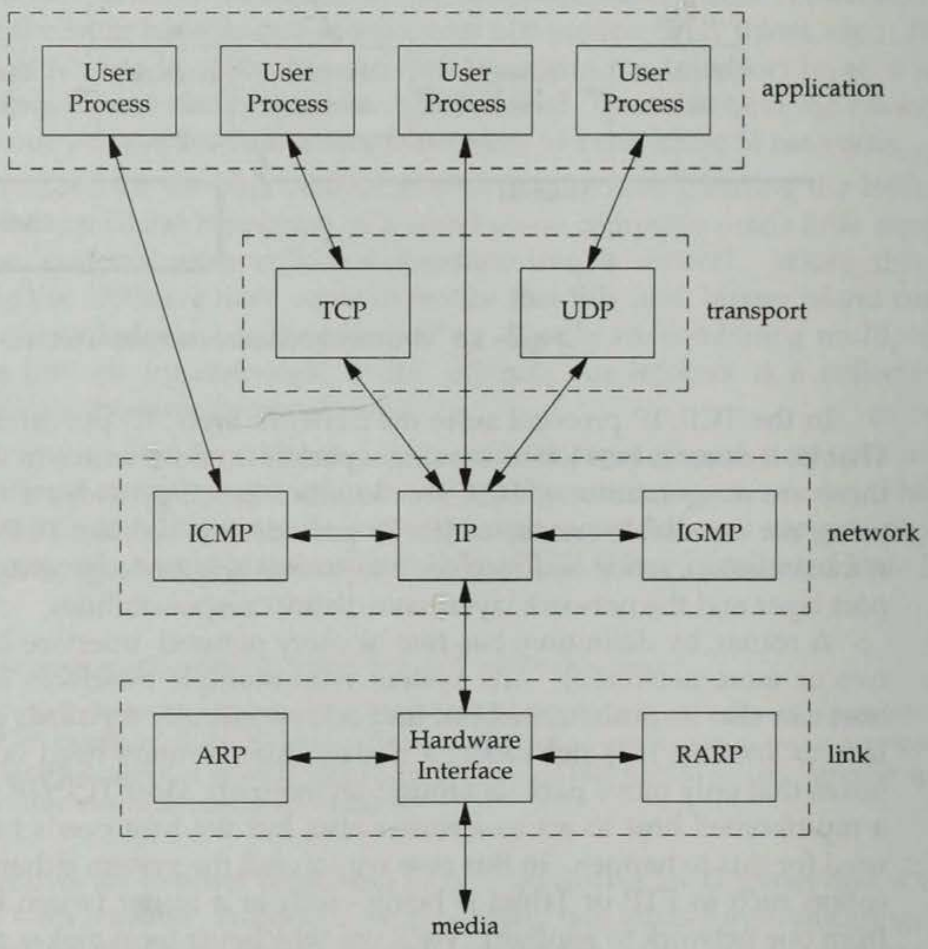


Figure 1.4 Various protocols at the different layers in the TCP/IP protocol suite.

TCP and UDP are the two predominant transport layer protocols. Both use IP as the network layer.

TCP provides a reliable transport layer, even though the service it uses (IP) is unreliable. Chapters 17 through 22 provide a detailed look at the operation of TCP. We then look at some TCP applications: Telnet and Rlogin in Chapter 26, FTP in Chapter 27, and SMTP in Chapter 28. The applications are normally user processes.

UDP sends and receives *datagrams* for applications. A datagram is a unit of information (i.e., a certain number of bytes of information that is specified by the sender) that travels from the sender to the receiver. Unlike TCP, however, UDP is unreliable. There is no guarantee that the datagram ever gets to its final destination. Chapter 11 looks at UDP, and then Chapter 14 (the Domain Name System), Chapter 15 (the Trivial File Transfer Protocol), and Chapter 16 (the Bootstrap Protocol) look at some applications that use UDP. SNMP (the Simple Network Management Protocol) also uses UDP, but since it deals with many of the other protocols, we save a discussion of it until Chapter 25.

IP is the main protocol at the network layer. It is used by both TCP and UDP. Every piece of TCP and UDP data that gets transferred around an internet goes through the IP layer at both end systems and at every intermediate router. In Figure 1.4 we also show an application accessing IP. This is rare, but possible. (Some older routing protocols were implemented this way. Also, it is possible to experiment with new transport layer protocols using this feature.) Chapter 3 looks at IP, but we save some of the details for later chapters where their discussion makes more sense. Chapters 9 and 10 look at how IP performs routing.

ICMP is an adjunct to IP. It is used by the IP layer to exchange error messages and other vital information with the IP layer in another host or router. Chapter 6 looks at ICMP in more detail. Although ICMP is used primarily by IP, it is possible for an application to also access it. Indeed we'll see that two popular diagnostic tools, Ping and Traceroute (Chapters 7 and 8), both use ICMP.

IGMP is the Internet Group Management Protocol. It is used with multicasting: sending a UDP datagram to multiple hosts. We describe the general properties of broadcasting (sending a UDP datagram to every host on a specified network) and multicasting in Chapter 12, and then describe IGMP itself in Chapter 13.

ARP (Address Resolution Protocol) and RARP (Reverse Address Resolution Protocol) are specialized protocols used only with certain types of network interfaces (such as Ethernet and token ring) to convert between the addresses used by the IP layer and the addresses used by the network interface. We examine these protocols in Chapters 4 and 5, respectively.

1.4 Internet Addresses

Every interface on an internet must have a unique *Internet address* (also called an *IP address*). These addresses are 32-bit numbers. Instead of using a flat address space such as 1, 2, 3, and so on, there is a structure to Internet addresses. Figure 1.5 shows the five different classes of Internet addresses.

These 32-bit addresses are normally written as four decimal numbers, one for each byte of the address. This is called *dotted-decimal* notation. For example, the class B address of the author's primary system is 140.252.13.33.

The easiest way to differentiate between the different classes of addresses is to look at the first number of a dotted-decimal address. Figure 1.6 shows the different classes, with the first number in boldface.

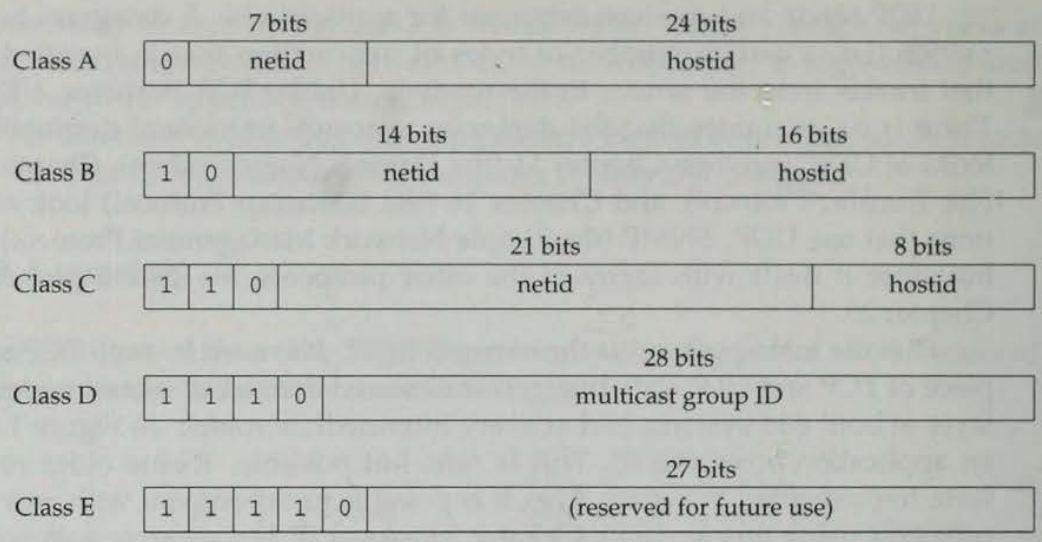


Figure 1.5 The five different classes of Internet addresses.

Class	Range
A	**0**.0.0.0 to **127**.255.255.255
B	**128**.0.0.0 to **191**.255.255.255
C	**192**.0.0.0 to **223**.255.255.255
D	**224**.0.0.0 to **239**.255.255.255
E	**240**.0.0.0 to **247**.255.255.255

Figure 1.6 Ranges for different classes of IP addresses.

It is worth reiterating that a multihomed host will have multiple IP addresses: one per interface.

Since every interface on an internet must have a unique IP address, there must be one central authority for allocating these addresses for networks connected to the worldwide Internet. That authority is the *Internet Network Information Center*, called the InterNIC. The InterNIC assigns only network IDs. The assignment of host IDs is up to the system administrator.

> Registration services for the Internet (IP addresses and DNS domain names) used to be handled by the NIC, at `nic.ddn.mil`. On April 1, 1993, the InterNIC was created. Now the NIC handles these requests only for the *Defense Data Network* (DDN). All other Internet users now use the InterNIC registration services, at `rs.internic.net`.
>
> There are actually three parts to the InterNIC: registration services (`rs.internic.net`), directory and database services (`ds.internic.net`), and information services (`is.internic.net`). See Exercise 1.8 for additional information on the InterNIC.

There are three types of IP addresses: *unicast* (destined for a single host), *broadcast* (destined for all hosts on a given network), and *multicast* (destined for a set of hosts that belong to a multicast group). Chapters 12 and 13 look at broadcasting and multicasting in more detail.

In Section 3.4 we'll extend our description of IP addresses to include subnetting, after describing IP routing. Figure 3.9 shows the special case IP addresses: host IDs and network IDs of all zero bits or all one bits.

1.5 The Domain Name System

Although the network interfaces on a host, and therefore the host itself, are known by IP addresses, humans work best using the *name* of a host. In the TCP/IP world the *Domain Name System* (DNS) is a distributed database that provides the mapping between IP addresses and hostnames. Chapter 14 looks into the DNS in detail.

For now we must be aware that any application can call a standard library function to look up the IP address (or addresses) corresponding to a given hostname. Similarly a function is provided to do the reverse lookup—given an IP address, look up the corresponding hostname.

Most applications that take a hostname as an argument also take an IP address. When we use the Telnet client in Chapter 4, for example, one time we specify a hostname and another time we specify an IP address.

1.6 Encapsulation

When an application sends data using TCP, the data is sent down the protocol stack, through each layer, until it is sent as a stream of bits across the network. Each layer adds information to the data by prepending headers (and sometimes adding trailer information) to the data that it receives. Figure 1.7 shows this process. The unit of data that TCP sends to IP is called a *TCP segment*. The unit of data that IP sends to the network interface is called an *IP datagram*. The stream of bits that flows across the Ethernet is called a *frame*.

The numbers at the bottom of the headers and trailer of the Ethernet frame in Figure 1.7 are the typical sizes of the headers in bytes. We'll have more to say about each of these headers in later sections.

A physical property of an Ethernet frame is that the size of its data must be between 46 and 1500 bytes. We'll encounter this minimum in Section 4.5 and we cover the maximum in Section 2.8.

> All the Internet standards and most books on TCP/IP use the term *octet* instead of byte. The use of this cute, but baroque term is historical, since much of the early work on TCP/IP was done on systems such as the DEC-10, which did not use 8-bit bytes. Since almost every current computer system uses 8-bit bytes, we'll use the term *byte* in this text.
>
> To be completely accurate in Figure 1.7 we should say that the unit of data passed between IP and the network interface is a *packet*. This packet can be either an IP datagram or a fragment of an IP datagram. We discuss fragmentation in detail in Section 11.5.

We could draw a nearly identical picture for UDP data. The only changes are that the unit of information that UDP passes to IP is called a *UDP datagram*, and the size of the UDP header is 8 bytes.

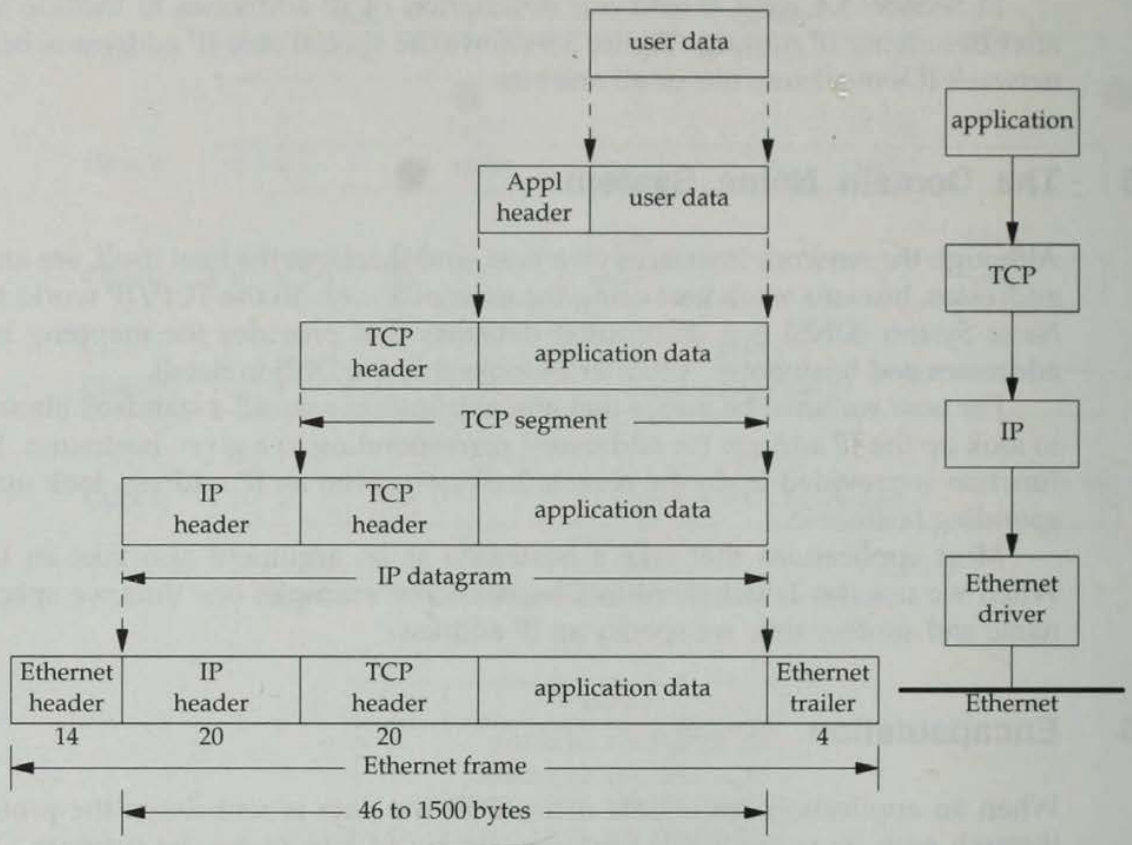


Figure 1.7 Encapsulation of data as it goes down the protocol stack.

Recall from Figure 1.4 (p. 6) that TCP, UDP, ICMP, and IGMP all send data to IP. IP must add some type of identifier to the IP header that it generates, to indicate the layer to which the data belongs. IP handles this by storing an 8-bit value in its header called the *protocol* field. A value of 1 is for ICMP, 2 is for IGMP, 6 indicates TCP, and 17 is for UDP.

Similarly, many different applications can be using TCP or UDP at any one time. The transport layer protocols store an identifier in the headers they generate to identify the application. Both TCP and UDP use 16-bit *port numbers* to identify applications. TCP and UDP store the source port number and the destination port number in their respective headers.

The network interface sends and receives frames on behalf of IP, ARP, and RARP. There must be some form of identification in the Ethernet header indicating which network layer protocol generated the data. To handle this there is a 16-bit frame type field in the Ethernet header.

1.7 Demultiplexing

When an Ethernet frame is received at the destination host it starts its way up the protocol stack and all the headers are removed by the appropriate protocol box. Each protocol box looks at certain identifiers in its header to determine which box in the next upper layer receives the data. This is called *demultiplexing*. Figure 1.8 shows how this takes place.

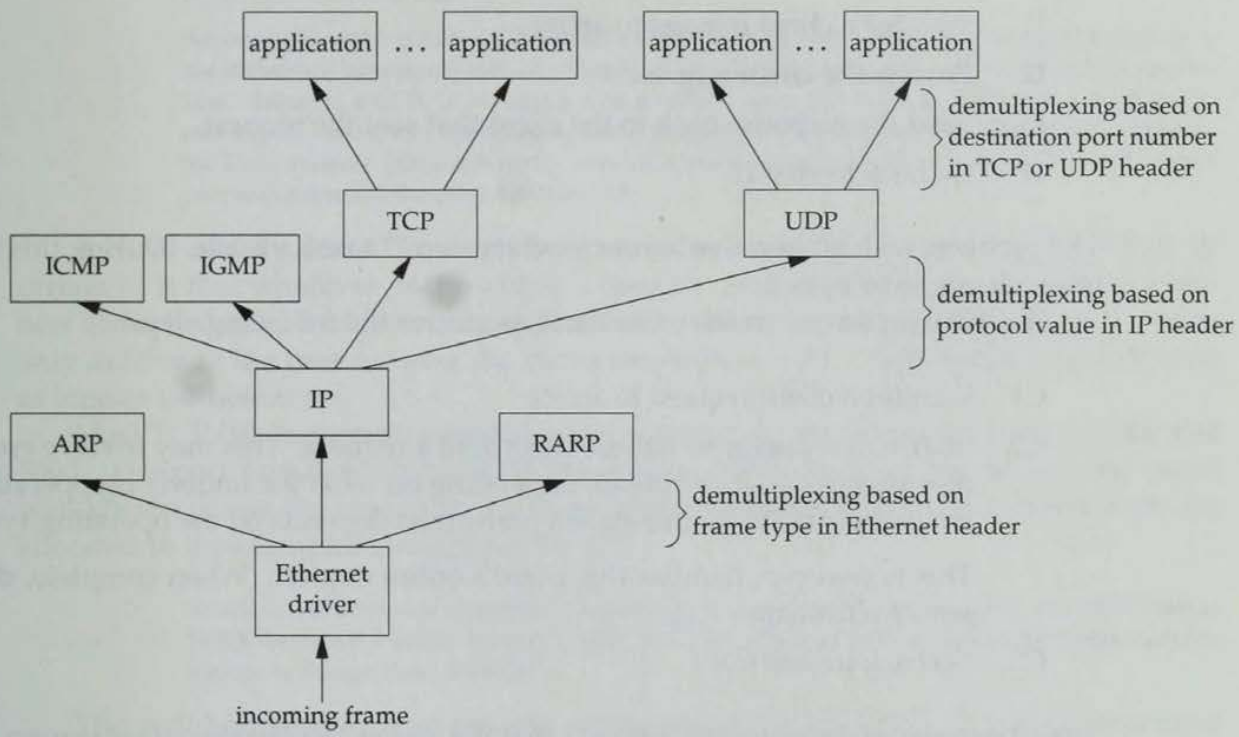


Figure 1.8 The demultiplexing of a received Ethernet frame.

Positioning the protocol boxes labeled "ICMP" and "IGMP" is always a challenge. In Figure 1.4 we showed them at the same layer as IP, because they really are adjuncts to IP. But here we show them above IP, to reiterate that ICMP messages and IGMP messages are encapsulated in IP datagrams.

We have a similar problem with the boxes "ARP" and "RARP." Here we show them above the Ethernet device driver because they both have their own Ethernet frame types, like IP datagrams. But in Figure 2.4 we'll show ARP as part of the Ethernet device driver, beneath IP, because that's where it logically fits.

Realize that these pictures of layered protocol boxes are not perfect.

When we describe TCP in detail we'll see that it really demultiplexes incoming segments using the destination port number, the source IP address, and the source port number.

1.8 Client–Server Model

Most networking applications are written assuming one side is the client and the other the server. The purpose of the application is for the server to provide some defined service for clients.

We can categorize servers into two classes: iterative or concurrent. An *iterative server* iterates through the following steps.

I1. Wait for a client request to arrive.

I2. Process the client request.

I3. Send the response back to the client that sent the request.

I4. Go back to step I1.

The problem with an iterative server is when step I2 takes a while. During this time no other clients are serviced.

A *concurrent server*, on the other hand, performs the following steps.

C1. Wait for a client request to arrive.

C2. Start a new server to handle this client's request. This may involve creating a new process, task, or thread, depending on what the underlying operating system supports. How this step is performed depends on the operating system.

This new server handles this client's entire request. When complete, this new server terminates.

C3. Go back to step C1.

The advantage of a concurrent server is that the server just spawns other servers to handle the client requests. Each client has, in essence, its own server. Assuming the operating system allows multiprogramming, multiple clients are serviced concurrently.

The reason we categorize servers, and not clients, is because a client normally can't tell whether it's talking to an iterative server or a concurrent server.

As a general rule, TCP servers are concurrent, and UDP servers are iterative, but there are a few exceptions. We'll look in detail at the impact of UDP on its servers in Section 11.12, and the impact of TCP on its servers in Section 18.11.

1.9 Port Numbers

We said that TCP and UDP identify applications using 16-bit port numbers. How are these port numbers chosen?

Servers are normally known by their *well-known* port number. For example, every TCP/IP implementation that provides an FTP server provides that service on TCP port

21. Every Telnet server is on TCP port 23. Every implementation of TFTP (the Trivial File Transfer Protocol) is on UDP port 69. Those services that can be provided by any implementation of TCP/IP have well-known port numbers between 1 and 1023. The well-known ports are managed by the *Internet Assigned Numbers Authority* (IANA).

> Until 1992 the well-known ports were between 1 and 255. Ports between 256 and 1023 were normally used by Unix systems for Unix-specific services—that is, services found on a Unix system, but probably not found on other operating systems. The IANA now manages the ports between 1 and 1023.
>
> An example of the difference between an Internet-wide service and a Unix-specific service is the difference between Telnet and Rlogin. Both allow us to login across a network to another host. Telnet is a TCP/IP standard with a well-known port number of 23 and can be implemented on almost any operating system. Rlogin, on the other hand, was originally designed for Unix systems (although many non-Unix systems now provide it also) so its well-known port was chosen in the early 1980s as 513.

A client usually doesn't care what port number it uses on its end. All it needs to be certain of is that whatever port number it uses be unique on its host. Client port numbers are called *ephemeral ports* (i.e., short lived). This is because a client typically exists only as long as the user running the client needs its service, while servers typically run as long as the host is up.

Most TCP/IP implementations allocate ephemeral port numbers between 1024 and 5000. The port numbers above 5000 are intended for other servers (those that aren't well known across the Internet). We'll see many examples of how ephemeral ports are allocated in the examples throughout the text.

> Solaris 2.2 is a notable exception. By default the ephemeral ports for TCP and UDP start at 32768. Section E.4 details the configuration options that can be modified by the system administrator to change these defaults.

The well-known port numbers are contained in the file `/etc/services` on most Unix systems. To find the port numbers for the Telnet server and the Domain Name System, we can execute

```
sun % grep telnet /etc/services
telnet    23/tcp                    says it uses TCP port 23

sun % grep domain /etc/services
domain    53/udp                    says it uses UDP port 53
domain    53/tcp                    and TCP port 53
```

Reserved Ports

Unix systems have the concept of *reserved ports*. Only a process with superuser privileges can assign itself a reserved port.

These port numbers are in the range of 1 to 1023, and are used by some applications (notably Rlogin, Section 26.2), as part of the authentication between the client and server.

1.10 Standardization Process

Who controls the TCP/IP protocol suite, approves new standards, and the like? There are four groups responsible for Internet technology.

1. The *Internet Society* (ISOC) is a professional society to facilitate, support, and promote the evolution and growth of the Internet as a global research communications infrastructure.
2. The *Internet Architecture Board* (IAB) is the technical oversight and coordination body. It is composed of about 15 international volunteers from various disciplines and serves as the final editorial and technical review board for the quality of Internet standards. The IAB falls under the ISOC.
3. The *Internet Engineering Task Force* (IETF) is the near-term, standards-oriented group, divided into nine areas (applications, routing and addressing, security, etc.). The IETF develops the specifications that become Internet standards. An additional *Internet Engineering Steering Group* (IESG) was formed to help the IETF chair.
4. The *Internet Research Task Force* (IRTF) pursues long-term research projects.

Both the IRTF and the IETF fall under the IAB. [Crocker 1993] provides additional details on the standardization process within the Internet, as well as some of its early history.

1.11 RFCs

All the official standards in the internet community are published as a *Request for Comment*, or *RFC*. Additionally there are lots of RFCs that are not official standards, but are published for informational purposes. The RFCs range in size from 1 page to almost 200 pages. Each is identified by a number, such as RFC 1122, with higher numbers for newer RFCs.

All the RFCs are available at no charge through electronic mail or using FTP across the Internet. Sending electronic mail as shown here:

```
To: rfc-info@ISI.EDU
Subject: getting rfcs

help: ways_to_get_rfcs
```

returns a detailed listing of various ways to obtain the RFCs.

The latest RFC index is always a starting point when looking for something. This index specifies when a certain RFC has been replaced by a newer RFC, and if a newer RFC updates some of the information in that RFC.

There are a few important RFCs.

1. The *Assigned Numbers RFC* specifies all the magic numbers and constants that are used in the Internet protocols. At the time of this writing the latest version

of this RFC is 1340 [Reynolds and Postel 1992]. All the Internet-wide well-known ports are listed here.

When this RFC is updated (it is normally updated at least yearly) the index listing for 1340 will indicate which RFC has replaced it.

2. The *Internet Official Protocol Standards*, currently RFC 1600 [Postel 1994]. This RFC specifies the state of standardization of the various Internet protocols. Each protocol has one of the following states of standardization: standard, draft standard, proposed standard, experimental, informational, or historic. Additionally each protocol has a requirement level: required, recommended, elective, limited use, or not recommended.

   Like the Assigned Numbers RFC, this RFC is also reissued regularly. Be sure you're reading the current copy.

3. The *Host Requirements RFCs*, 1122 and 1123 [Braden 1989a, 1989b]. RFC 1122 handles the link layer, network layer, and transport layer, while RFC 1123 handles the application layer. These two RFCs make numerous corrections and interpretations of the important earlier RFCs, and are often the starting point when looking at any of the finer details of a given protocol. They list the features and implementation details of the protocols as either "must," "should," "may," "should not," or "must not."

   [Borman 1993b] provides a practical look at these two RFCs, and RFC 1127 [Braden 1989c] provides an informal summary of the discussions and conclusions of the working group that developed the Host Requirements RFCs.

4. The *Router Requirements RFC*. The official version of this is RFC 1009 [Braden and Postel 1987], but a new version is nearing completion [Almquist 1993]. This is similar to the host requirements RFCs, but specifies the unique requirements of routers.

1.12 Standard, Simple Services

There are a few standard, simple services that almost every implementation provides. We'll use some of these servers throughout the text, usually with the Telnet client. Figure 1.9 describes these services. We can see from this figure that when the same service is provided using both TCP and UDP, both port numbers are normally chosen to be the same.

> If we examine the port numbers for these standard services and other standard TCP/IP services (Telnet, FTP, SMTP, etc.), most are odd numbers. This is historical as these port numbers are derived from the NCP port numbers. (NCP, the Network Control Protocol, preceded TCP as a transport layer protocol for the ARPANET.) NCP was simplex, not full-duplex, so each application required two connections, and an even–odd pair of port numbers was reserved for each application. When TCP and UDP became the standard transport layers, only a single port number was needed per application, so the odd port numbers from NCP were used.

Name	TCP port	UDP port	RFC	Description
echo	7	7	862	Server returns whatever the client sends.
discard	9	9	863	Server discards whatever the client sends.
daytime	13	13	867	Server returns the time and date in a human-readable format.
chargen	19	19	864	TCP server sends a continual stream of characters, until the connection is terminated by the client. UDP server sends a datagram containing a random number of characters each time the client sends a datagram.
time	37	37	868	Server returns the time as a 32-bit binary number. This number represents the number of seconds since midnight January 1, 1900, UTC.

Figure 1.9 Standard, simple services provided by most implementations.

1.13 The Internet

In Figure 1.3 we showed an *internet* composed of two networks—an Ethernet and a token ring. In Sections 1.4 and 1.9 we talked about the worldwide *Internet* and the need to allocate IP addresses centrally (the InterNIC) and the well-known port numbers (the IANA). The word *internet* means different things depending on whether it's capitalized or not.

The lowercase *internet* means multiple networks connected together, using a common protocol suite. The uppercase *Internet* refers to the collection of hosts (over one million) around the world that can communicate with each other using TCP/IP. While the Internet is an internet, the reverse is not true.

1.14 Implementations

The de facto standard for TCP/IP implementations is the one from the Computer Systems Research Group at the University of California at Berkeley. Historically this has been distributed with the 4.x BSD system (Berkeley Software Distribution), and with the "BSD Networking Releases." This source code has been the starting point for many other implementations.

Figure 1.10 shows a chronology of the various BSD releases, indicating the important TCP/IP features. The BSD Networking Releases shown on the left side are publicly available source code releases containing all of the networking code: both the protocols themselves and many of the applications and utilities (such as Telnet and FTP).

> As of this writing there is an unresolved legal battle that will determine if and when a Net/3 release occurs.

Throughout the text we'll use the term *Berkeley-derived implementation* to refer to vendor implementations such as SunOS 4.x, SVR4, and AIX 3.2 that were originally

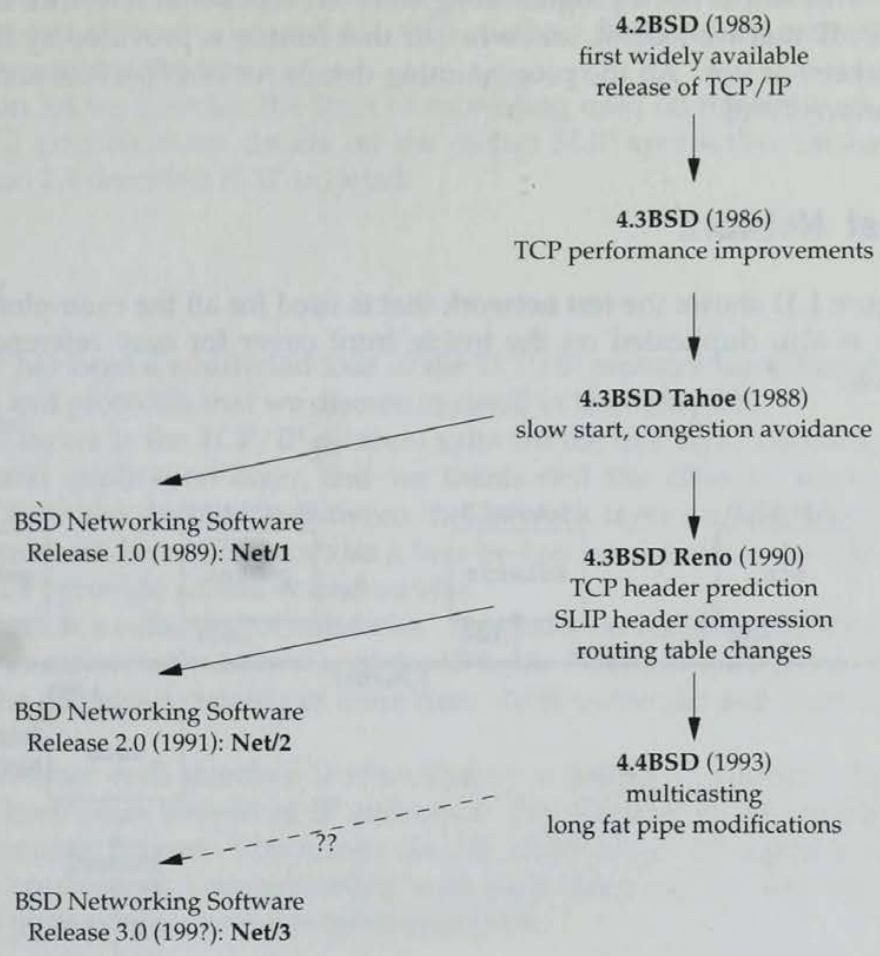


Figure 1.10 Various BSD releases with important TCP/IP features.

developed from the Berkeley sources. These implementations have much in common, often including the same bugs!

Much of the original research in the Internet is still being applied to the Berkeley system—new congestion control algorithms (Section 21.7), multicasting (Section 12.4), "long fat pipe" modifications (Section 24.3), and the like.

1.15 Application Programming Interfaces

Two popular *application programming interfaces* (APIs) for applications using the TCP/IP protocols are called *sockets* and *TLI* (Transport Layer Interface). The former is sometimes called "Berkeley sockets," indicating where it was originally developed. The latter, originally developed by AT&T, is sometimes called *XTI* (X/Open Transport Interface), recognizing the work done by X/Open, an international group of computer vendors that produce their own set of standards. XTI is effectively a superset of TLI.

This text is not a programming text, but occasional reference is made to features of TCP/IP that we look at, and whether that feature is provided by the most popular API (sockets) or not. All the programming details for both sockets and TLI are available in [Stevens 1990].

1.16 Test Network

Figure 1.11 shows the test network that is used for all the examples in the text. This figure is also duplicated on the inside front cover for easy reference while reading the book.

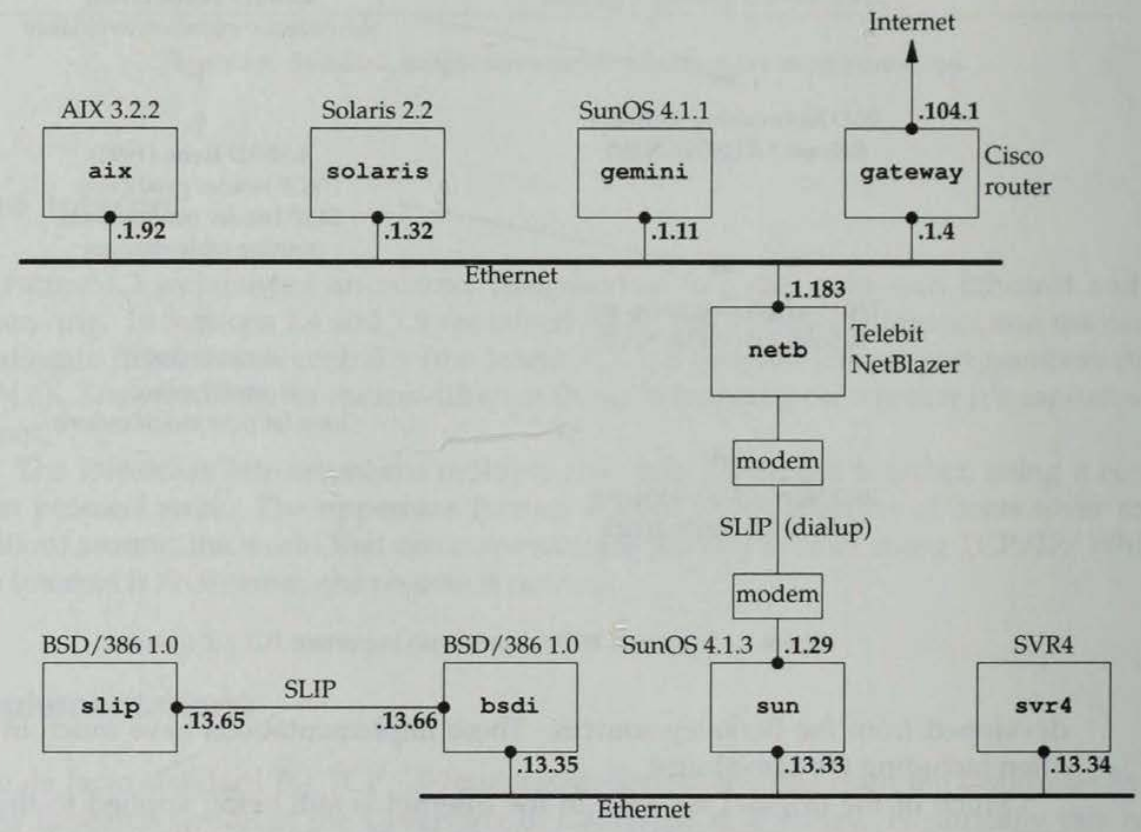


Figure 1.11 Test network used for all the examples in the text. All IP addresses begin with 140.252.

Most of the examples are run on the lower four systems in this figure (the author's subnet). All the IP addresses in this figure belong to the class B network ID 140.252. All the hostnames belong to the `.tuc.noao.edu` domain. (`noao` stands for "National Optical Astronomy Observatories" and `tuc` stands for Tucson.) For example, the lower right system has a complete hostname of `svr4.tuc.noao.edu` and an IP address of 140.252.13.34. The notation at the top of each box is the operating system running on that system. This collection of systems and networks provides hosts and routers running a variety of TCP/IP implementations.

It should be noted that there are many more networks and hosts in the `noao.edu` domain than we show in Figure 1.11. All we show here are the systems that we'll encounter throughout the text.

In Section 3.4 we describe the form of subnetting used on this network, and in Section 4.6 we'll provide more details on the dialup SLIP connection between `sun` and `netb`. Section 2.4 describes SLIP in detail.

1.17 Summary

This chapter has been a whirlwind tour of the TCP/IP protocol suite, introducing many of the terms and protocols that we discuss in detail in later chapters.

The four layers in the TCP/IP protocol suite are the link layer, network layer, transport layer, and application layer, and we mentioned the different responsibilities of each. In TCP/IP the distinction between the network layer and the transport layer is critical: the network layer (IP) provides a hop-by-hop service while the transport layers (TCP and UDP) provide an end-to-end service.

An internet is a collection of networks. The common building block for an internet is a router that connects the networks at the IP layer. The capital-I Internet is an internet that spans the globe and consists of more than 10,000 networks and more than one million computers.

On an internet each interface is identified by a unique IP address, although users tend to use hostnames instead of IP addresses. The Domain Name System provides a dynamic mapping between hostnames and IP addresses. Port numbers are used to identify the applications communicating with each other and we said that servers use well-known ports while clients use ephemeral ports.

Exercises

1.1 Calculate the maximum number of class A, B, and C network IDs.

1.2 Fetch the file `nsfnet/statistics/history.netcount` using anonymous FTP (Section 27.3) from the host `nic.merit.edu`. This file contains the number of domestic and foreign networks announced to the NSFNET infrastructure. Plot these values with the year on the x-axis and a logarithmic y-axis with the total number of networks. The maximum value for the y-axis should be the value calculated in the previous exercise. If the data shows a visual trend, extrapolate the values to estimate when the current addressing scheme will run out of network IDs. (Section 3.10 talks about proposals to correct this problem.)

1.3 Obtain a copy of the Host Requirements RFC [Braden 1989a] and look up the *robustness principle* that applies to every layer of the TCP/IP protocol suite. What is the reference for this principle?

1.4 Obtain a copy of the latest Assigned Numbers RFC. What is the well-known port for the "quote of the day" protocol? Which RFC defines the protocol?

1.5 If you have an account on a host that is connected to a TCP/IP internet, what is its primary IP address? Is the host connected to the worldwide Internet? Is it multihomed?

1.6 Obtain a copy of RFC 1000 to learn where the term RFC originated.

1.7 Contact the Internet Society, `isoc@isoc.org` or +1 703 648 9888, to find out about joining.

1.8 Fetch the file `about-internic/information-about-the-internic` using anonymous FTP from the host `is.internic.net`.

2

Link Layer

2.1 Introduction

From Figure 1.4 (p. 6) we see that the purpose of the link layer in the TCP/IP protocol suite is to send and receive (1) IP datagrams for the IP module, (2) ARP requests and replies for the ARP module, and (3) RARP requests and replies for the RARP module. TCP/IP supports many different link layers, depending on the type of networking hardware being used: Ethernet, token ring, FDDI (Fiber Distributed Data Interface), RS-232 serial lines, and the like.

In this chapter we'll look at some of the details involved in the Ethernet link layer, two specialized link layers for serial interfaces (SLIP and PPP), and the loopback driver that's part of most implementations. Ethernet and SLIP are the link layers used for most of the examples in the book. We also talk about the MTU (Maximum Transmission Unit), a characteristic of the link layer that we encounter numerous times in the remaining chapters. We also show some calculations of how to choose the MTU for a serial line.

2.2 Ethernet and IEEE 802 Encapsulation

The term *Ethernet* generally refers to a standard published in 1982 by Digital Equipment Corp., Intel Corp., and Xerox Corp. It is the predominant form of local area network technology used with TCP/IP today. It uses an access method called CSMA/CD, which stands for Carrier Sense, Multiple Access with Collision Detection. It operates at 10 Mbits/sec and uses 48-bit addresses.

A few years later the IEEE (Institute of Electrical and Electronics Engineers) 802 Committee published a sightly different set of standards. 802.3 covers an entire set of

CSMA/CD networks, 802.4 covers token bus networks, and 802.5 covers token ring networks. Common to all three of these is the 802.2 standard that defines the logical link control (LLC) common to many of the 802 networks. Unfortunately the combination of 802.2 and 803.3 defines a different frame format from true Ethernet. ([Stallings 1987] covers all the details of these IEEE 802 standards.)

In the TCP/IP world, the encapsulation of IP datagrams is defined in RFC 894 [Hornig 1984] for Ethernets and in RFC 1042 [Postel and Reynolds 1988] for IEEE 802 networks. The Host Requirements RFC requires that every Internet host connected to a 10 Mbits/sec Ethernet cable:

1. Must be able to send and receive packets using RFC 894 (Ethernet) encapsulation.
2. Should be able to receive RFC 1042 (IEEE 802) packets intermixed with RFC 894 packets.
3. May be able to send packets using RFC 1042 encapsulation. If the host can send both types of packets, the type of packet sent must be configurable and the configuration option must default to RFC 894 packets.

RFC 894 encapsulation is most commonly used. Figure 2.1 shows the two different forms of encapsulation. The number below each box in the figure is the size of that box in bytes.

Both frame formats use 48-bit (6-byte) destination and source addresses. (802.3 allows 16-bit addresses to be used, but 48-bit addresses are normal.) These are what we call *hardware addresses* throughout the text. The ARP and RARP protocols (Chapters 4 and 5) map between the 32-bit IP addresses and the 48-bit hardware addresses.

The next 2 bytes are different in the two frame formats. The 802 *length* field says how many bytes follow, up to but not including the *CRC* at the end. The Ethernet *type* field identifies the type of data that follows. In the 802 frame the same *type* field occurs later in the SNAP (Sub-network Access Protocol) header. Fortunately none of the valid 802 *length* values is the same as the Ethernet *type* values, making the two frame formats distinguishable.

In the Ethernet frame the data immediately follows the *type* field, while in the 802 frame format 3 bytes of 802.2 LLC and 5 bytes of 802.2 SNAP follow. The *DSAP* (Destination Service Access Point) and *SSAP* (Source Service Access Point) are both set to `0xaa`. The *ctrl* field is set to 3. The next 3 bytes, the *org code* are all 0. Following this is the same 2-byte *type* field that we had with the Ethernet frame format. (Additional type field values are given in RFC 1340 [Reynolds and Postel 1992].)

The *CRC* field is a cyclic redundancy check (a checksum) that detects errors in the rest of the frame. (This is also called the *FCS* or frame check sequence.)

There is a minimum size for 802.3 and Ethernet frames. This minimum requires that the data portion be at least 38 bytes for 802.3 or 46 bytes for Ethernet. To handle this, pad bytes are inserted to assure that the frame is long enough. We'll encounter this minimum when we start watching packets on the wire.

In this text we'll display the Ethernet encapsulation when we need to, because this is the most commonly used form of encapsulation.

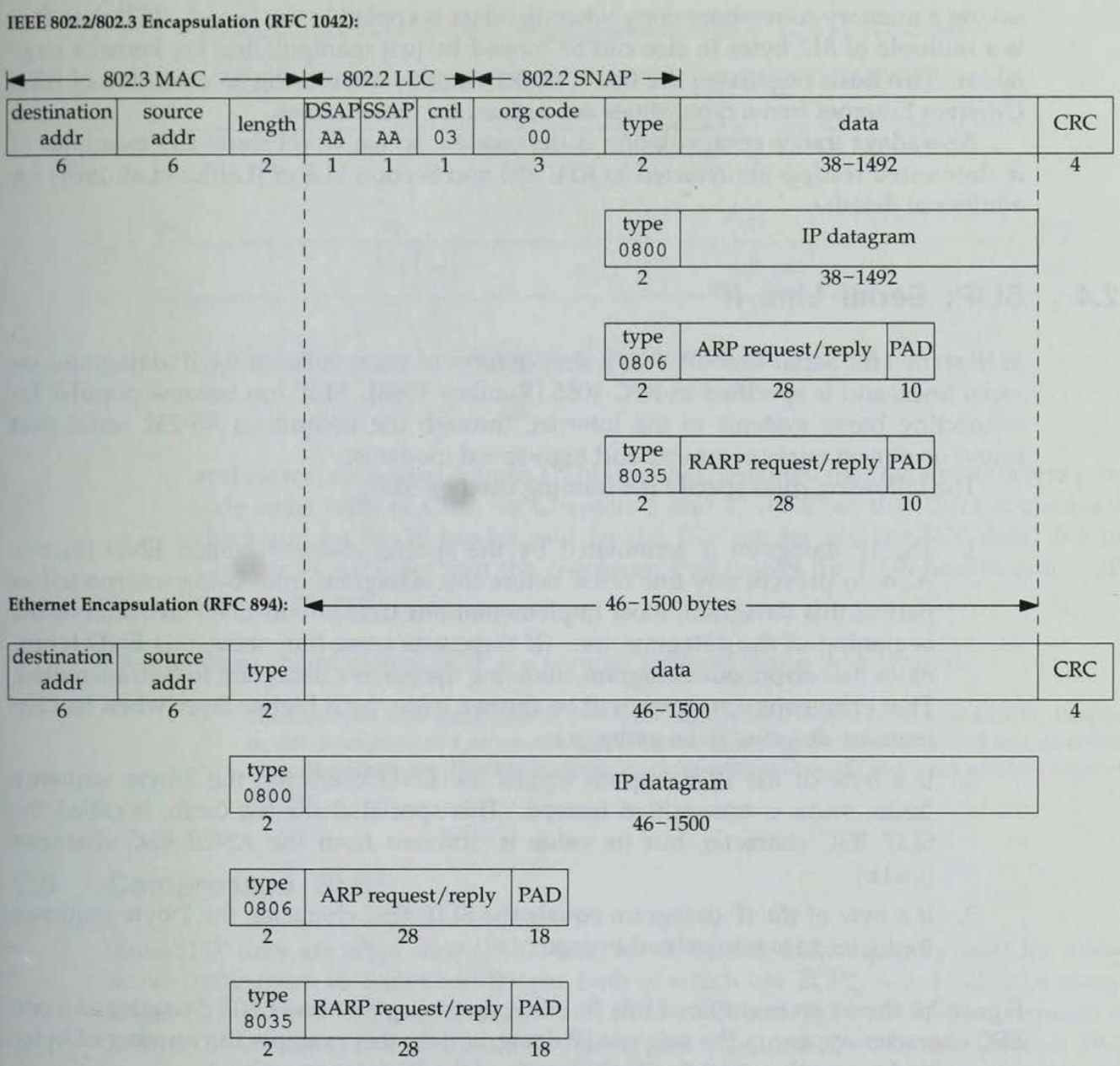


Figure 2.1 IEEE 802.2/802.3 encapsulation (RFC 1042) and Ethernet encapsulation (RFC 894).

2.3 Trailer Encapsulation

RFC 893 [Leffler and Karels 1984] describes another form of encapsulation used on Ethernets, called *trailer encapsulation*. It was an experiment with early BSD systems on DEC VAXes that improved performance by rearranging the order of the fields in the IP datagram. The variable-length fields at the beginning of the data portion of the Ethernet frame (the IP header and the TCP header) were moved to the end (right before the CRC). This allows the data portion of the frame to be mapped to a hardware page,

saving a memory-to-memory copy when the data is copied in the kernel. TCP data that is a multiple of 512 bytes in size can be moved by just manipulating the kernel's page tables. Two hosts negotiated the use of trailer encapsulation using an extension of ARP. Different Ethernet frame type values are defined for these frames.

Nowadays trailer encapsulation is deprecated, so we won't show any examples of it. Interested readers are referred to RFC 893 and Section 11.8 of [Leffler et al. 1989] for additional details.

2.4 SLIP: Serial Line IP

SLIP stands for Serial Line IP. It is a simple form of encapsulation for IP datagrams on serial lines, and is specified in RFC 1055 [Romkey 1988]. SLIP has become popular for connecting home systems to the Internet, through the ubiquitous RS-232 serial port found on almost every computer and high-speed modems.

The following rules specify the framing used by SLIP.

1. The IP datagram is terminated by the special character called END (`0xc0`). Also, to prevent any line noise before this datagram from being interpreted as part of this datagram, most implementations transmit an END character at the beginning of the datagram too. (If there was some line noise, this END terminates that erroneous datagram, allowing the current datagram to be transmitted. That erroneous datagram will be thrown away by a higher layer when its contents are detected to be garbage.)
2. If a byte of the IP datagram equals the END character, the 2-byte sequence `0xdb`, `0xdc` is transmitted instead. This special character, `0xdb`, is called the SLIP ESC character, but its value is different from the ASCII ESC character (`0x1b`).
3. If a byte of the IP datagram equals the SLIP ESC character, the 2-byte sequence `0xdb`, `0xdd` is transmitted instead.

Figure 2.2 shows an example of this framing, assuming that one END character and one ESC character appear in the original IP datagram. In this example the number of bytes transmitted across the serial line is the length of the IP datagram plus 4.

SLIP is a simple framing method. It has some deficiencies that are worth noting.

1. Each end must know the other's IP address. There is no method for one end to inform the other of its IP address.
2. There is no type field (similar to the frame type field in Ethernet frames). If a serial line is used for SLIP, it can't be used for some other protocol at the same time.
3. There is no checksum added by SLIP (similar to the CRC field in Ethernet frames). If a noisy phone line corrupts a datagram being transferred by SLIP, it's up to the higher layers to detect this. (Alternately, newer modems can detect

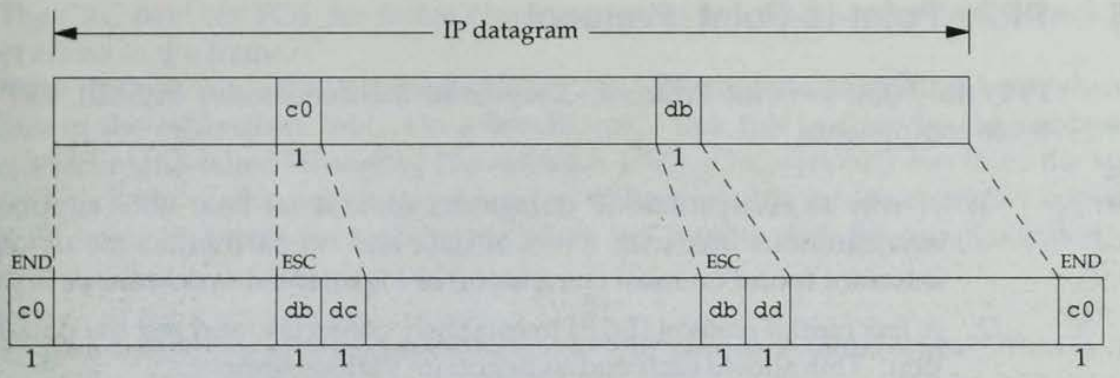


Figure 2.2 SLIP encapsulation.

and correct corrupted frames.) This makes it essential that the upper layers provide some form of CRC. In Chapters 3 and 17 we'll see that there is always a checksum for the IP header, and for the TCP header and the TCP data. But in Chapter 11 we'll see that the checksum that covers the UDP header and UDP data is optional.

Despite these shortcomings, SLIP is a popular protocol that is widely used.

The history of SLIP dates back to 1984 when Rick Adams implemented it in 4.2BSD. Despite its self-description as a nonstandard, it is becoming more popular as the speed and reliability of modems increase. Publicly available implementations abound, and many vendors support it today.

2.5 Compressed SLIP

Since SLIP lines are often slow (19200 bits/sec or below) and frequently used for interactive traffic (such as Telnet and Rlogin, both of which use TCP), there tend to be many small TCP packets exchanged across a SLIP line. To carry 1 byte of data requires a 20-byte IP header and a 20-byte TCP header, an overhead of 40 bytes. (Section 19.2 shows the flow of these small packets when a simple command is typed during an Rlogin session.)

Recognizing this performance drawback, a newer version of SLIP, called CSLIP (for compressed SLIP), is specified in RFC 1144 [Jacobson 1990a]. CSLIP normally reduces the 40-byte header to 3 or 5 bytes. It maintains the state of up to 16 TCP connections on each end of the CSLIP link and knows that some of the fields in the two headers for a given connection normally don't change. Of the fields that do change, most change by a small positive amount. These smaller headers greatly improve the interactive response time.

Most SLIP implementations today support CSLIP. Both SLIP links on the author's subnet (see inside front cover) are CSLIP links.

2.6 PPP: Point-to-Point Protocol

PPP, the Point-to-Point Protocol, corrects all the deficiencies in SLIP. PPP consists of three components.

1. A way to encapsulate IP datagrams on a serial link. PPP supports either an asynchronous link with 8 bits of data and no parity (i.e., the ubiquitous serial interface found on most computers) or bit-oriented synchronous links.
2. A *link control protocol* (LCP) to establish, configure, and test the data-link connection. This allows each end to negotiate various options.
3. A family of *network control protocols* (NCPs) specific to different network layer protocols. RFCs currently exist for IP, the OSI network layer, DECnet, and AppleTalk. The IP NCP, for example, allows each end to specify if it can perform header compression, similar to CSLIP. (The acronym NCP was also used for the predecessor to TCP.)

RFC 1548 [Simpson 1993] specifies the encapsulation method and the link control protocol. RFC 1332 [McGregor 1992] specifies the network control protocol for IP.

The format of the PPP frames was chosen to look like the ISO HDLC standard (high-level data link control). Figure 2.3 shows the format of PPP frames.

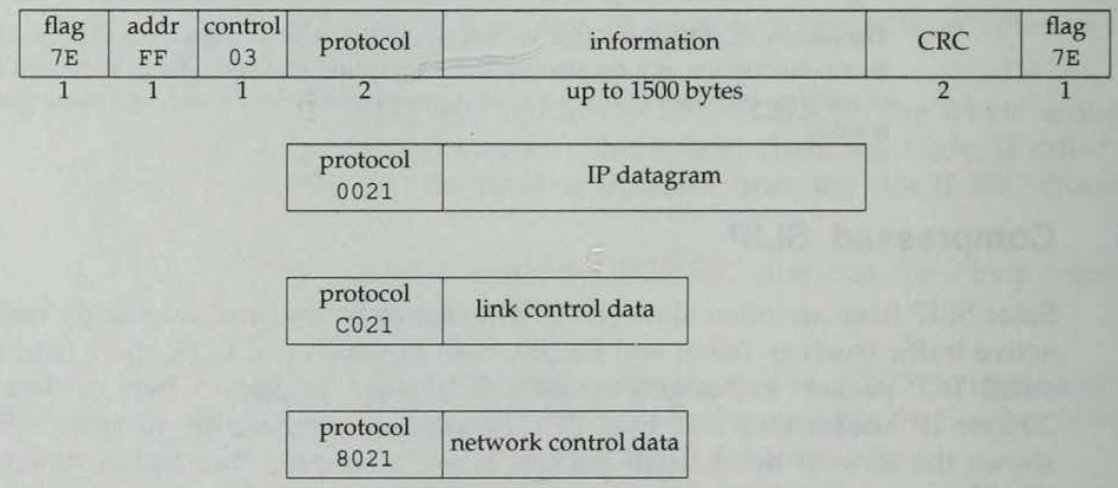


Figure 2.3 Format of PPP frames.

Each frame begins and ends with a *flag* byte whose value is `0x7e`. This is followed by an *address* byte whose value is always `0xff`, and then a *control* byte, with a value of `0x03`.

Next comes the *protocol* field, similar in function to the Ethernet *type* field. A value of `0x0021` means the *information* field is an IP datagram, a value of `0xc021` means the *information* field is link control data, and a value of `0x8021` is for network control data.

The *CRC* field (or FCS, for frame check sequence) is a cyclic redundancy check, to detect errors in the frame.

Since the byte value `0x7e` is the *flag* character, PPP needs to escape this byte when it appears in the *information* field. On a synchronous link this is done by the hardware using a technique called *bit stuffing* [Tanenbaum 1989]. On asynchronous links the special byte `0x7d` is used as an escape character. Whenever this escape character appears in a PPP frame, the next character in the frame has had its sixth bit complemented, as follows:

1. The byte `0x7e` is transmitted as the 2-byte sequence `0x7d, 0x5e`. This is the escape of the *flag* byte.
2. The byte `0x7d` is transmitted as the 2-byte sequence `0x7d, 0x5d`. This is the escape of the escape byte.
3. By default, a byte with a value less than `0x20` (i.e., an ASCII control character) is also escaped. For example, the byte `0x01` is transmitted as the 2-byte sequence `0x7d, 0x21`. (In this case the complement of the sixth bit turns the bit on, whereas in the two previous examples the complement turned the bit off.)

   The reason for doing this is to prevent these bytes from appearing as ASCII control characters to the serial driver on either host, or to the modems, which sometimes interpret these control characters specially. It is also possible to use the link control protocol to specify which, if any, of these 32 values must be escaped. By default, all 32 are escaped.

Since PPP, like SLIP, is often used across slow serial links, reducing the number of bytes per frame reduces the latency for interactive applications. Using the link control protocol, most implementations negotiate to omit the constant *flag* and *address* fields and to reduce the size of the *protocol* field from 2 bytes to 1 byte. If we then compare the framing overhead in a PPP frame, versus the 2-byte framing overhead in a SLIP frame (Figure 2.2), we see that PPP adds three additional bytes: 1 byte for the *protocol* field, and 2 bytes for the *CRC*. Additionally, using the IP network control protocol, most implementations then negotiate to use Van Jacobson header compression (identical to CSLIP compression) to reduce the size of the IP and TCP headers.

In summary, PPP provides the following advantages over SLIP: (1) support for multiple protocols on a single serial line, not just IP datagrams, (2) a cyclic redundancy check on every frame, (3) dynamic negotiation of the IP address for each end (using the IP network control protocol), (4) TCP and IP header compression similar to CSLIP, and (5) a link control protocol for negotiating many data-link options. The price we pay for all these features is 3 bytes of additional overhead per frame, a few frames of negotiation when the link is established, and a more complex implementation.

Despite all the added benefits of PPP over SLIP, today there are more SLIP users than PPP users. As implementations become more widely available, and as vendors start to support PPP, it should (eventually) replace SLIP.

2.7 Loopback Interface

Most implementations support a *loopback interface* that allows a client and server on the same host to communicate with each other using TCP/IP. The class A network ID 127 is reserved for the loopback interface. By convention, most systems assign the IP address of 127.0.0.1 to this interface and assign it the name `localhost`. An IP datagram sent to the loopback interface must not appear on any network.

Although we could imagine the transport layer detecting that the other end is the loopback address, and short circuiting some of the transport layer logic and all of the network layer logic, most implementations perform complete processing of the data in the transport layer and network layer, and only loop the IP datagram back to itself when the datagram leaves the bottom of the network layer.

Figure 2.4 shows a simplified diagram of how the loopback interface processes IP datagrams.

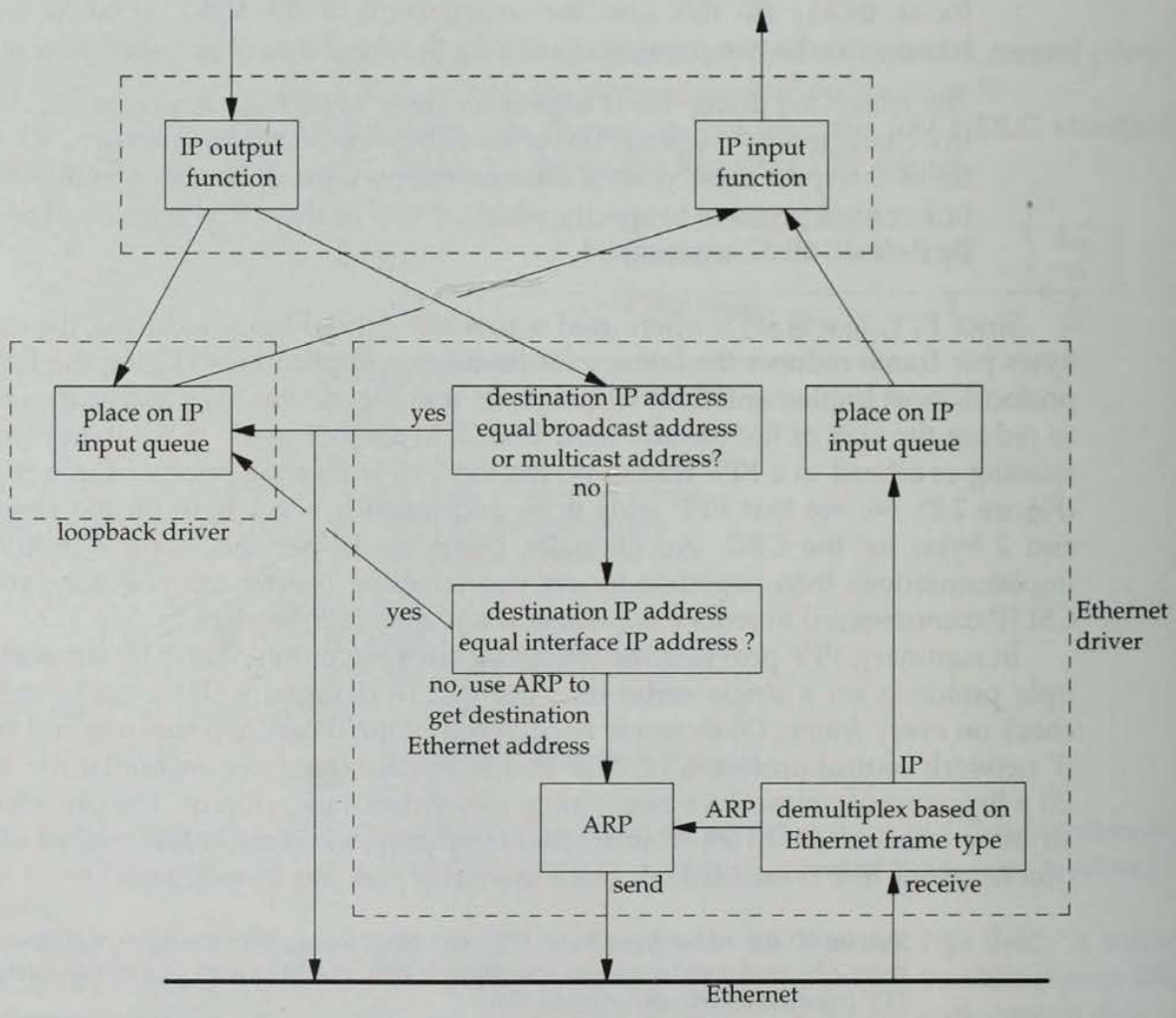


Figure 2.4 Processing of IP datagrams by loopback interface.

The key points to note in this figure are as follows:

1. Everything sent to the loopback address (normally 127.0.0.1) appears as IP input.
2. Datagrams sent to a broadcast address or a multicast address are copied to the loopback interface and sent out on the Ethernet. This is because the definition of broadcasting and multicasting (Chapter 12) includes the sending host.
3. Anything sent to one of the host's own IP addresses is sent to the loopback interface.

While it may seem inefficient to perform all the transport layer and IP layer processing of the loopback data, it simplifies the design because the loopback interface appears as just another link layer to the network layer. The network layer passes a datagram to the loopback interface like any other link layer, and it happens that the loopback interface then puts the datagram back onto IP's input queue.

Another implication of Figure 2.4 is that IP datagrams sent to the one of the host's own IP addresses normally do not appear on the corresponding network. For example, on an Ethernet, normally the packet is not transmitted and then read back. Comments in some of the BSD Ethernet device drivers indicate that many Ethernet interface cards are not capable of reading their own transmissions. Since a host must process IP datagrams that it sends to itself, handling these packets as shown in Figure 2.4 is the simplest way to accomplish this.

> The 4.4BSD implementation defines the variable `useloopback` and initializes it to 1. If this variable is set to 0, however, the Ethernet driver sends local packets onto the network instead of sending them to the loopback driver. This may or may not work, depending on your Ethernet interface card and device driver.

2.8 MTU

As we can see from Figure 2.1, there is a limit on the size of the frame for both Ethernet encapsulation and 802.3 encapsulation. This limits the number of bytes of data to 1500 and 1492, respectively. This characteristic of the link layer is called the *MTU*, its maximum transmission unit. Most types of networks have an upper limit.

If IP has a datagram to send, and the datagram is larger than the link layer's MTU, IP performs *fragmentation*, breaking the datagram up into smaller pieces (fragments), so that each fragment is smaller than the MTU. We discuss IP fragmentation in Section 11.5.

Figure 2.5 lists some typical MTU values, taken from RFC 1191 [Mogul and Deering 1990]. The listed MTU for a point-to-point link (e.g., SLIP or PPP) is not a physical characteristic of the network media. Instead it is a logical limit to provide adequate response time for interactive use. In the Section 2.10 we'll see where this limit comes from.

In Section 3.8 we'll use the `ifconfig` command to print the MTU of an interface.

Network	MTU (bytes)
Hyperchannel	65535
16 Mbits/sec token ring (IBM)	17914
4 Mbits/sec token ring (IEEE 802.5)	4464
FDDI	4352
Ethernet	1500
IEEE 802.3/802.2	1492
X.25	576
Point-to-point (low delay)	296

Figure 2.5 Typical maximum transmission units (MTUs).

2.9 Path MTU

When two hosts on the same network are communicating with each other, it is the MTU of the network that is important. But when two hosts are communicating across multiple networks, each link can have a different MTU. The important numbers are not the MTUs of the two networks to which the two hosts connect, but rather the smallest MTU of any data link that packets traverse between the two hosts. This is called the *path MTU*.

The path MTU between any two hosts need not be constant. It depends on the route being used at any time. Also, routing need not be symmetric (the route from A to B may not be the reverse of the route from B to A), hence the path MTU need not be the same in the two directions.

RFC 1191 [Mogul and Deering 1990] specifies the "path MTU discovery mechanism," a way to determine the path MTU at any time. We'll see how this mechanism operates after we've described ICMP and IP fragmentation. In Section 11.6 we'll examine the ICMP unreachable error that is used with this discovery mechanism and in Section 11.7 we'll show a version of the `traceroute` program that uses this mechanism to determine the path MTU to a destination. Sections 11.8 and 24.2 show how UDP and TCP operate when the implementation supports path MTU discovery.

2.10 Serial Line Throughput Calculations

If the line speed is 9600 bits/sec, with 8 bits per byte, plus 1 start bit and 1 stop bit, the line speed is 960 bytes/sec. Transferring a 1024-byte packet at this speed takes 1066 ms. If we're using the SLIP link for an interactive application, along with an application such as FTP that sends or receives 1024-byte packets, we have to wait, on the average, half of this time (533 ms) to send our interactive packet.

This assumes that our interactive packet will be sent across the link before any further "big" packets. Most SLIP implementations do provide this type-of-service queueing, placing interactive traffic ahead of bulk data traffic. The interactive traffic is normally Telnet, Rlogin, and the control portion (the user commands, not the data) of FTP.

This type of service queueing is imperfect. It cannot affect noninteractive traffic that is already queued downstream (e.g., at the serial driver). Also newer modems have large buffers so noninteractive traffic may already be buffered in the modem.

Waiting 533 ms is unacceptable for interactive response. Human factors studies have found that an interactive response time longer than 100–200 ms is perceived as bad [Jacobson 1990a]. This is the round-trip time for an interactive packet to be sent and something to be returned (normally a character echo).

Reducing the MTU of the SLIP link to 256 means the maximum amount of time the link can be busy with a single frame is 266 ms, and half of this (our average wait) is 133 ms. This is better, but still not perfect. The reason we choose this value (as compared to 64 or 128) is to provide good utilization of the line for bulk data transfers (such as large file transfers). Assuming a 5-byte CSLIP header, 256 bytes of data in a 261-byte frame gives 98.1% of the line to data and 1.9% to headers, which is good utilization. Reducing the MTU below 256 reduces the maximum throughput that we can achieve for bulk data transfers.

The MTU value listed in Figure 2.5, 296 for a point-to-point link, assumes 256 bytes of data and the 40-byte TCP and IP headers. Since the MTU is a value that IP queries the link layer for, the value must include the normal TCP and IP headers. This is how IP makes its fragmentation decision. IP knows nothing about the header compression that CSLIP performs.

Our average wait calculation (one-half the time required to transfer a maximum sized frame) only applies when a SLIP link (or PPP link) is used for both interactive traffic and bulk data transfer. When only interactive traffic is being exchanged, 1 byte of data in each direction (assuming 5-byte compressed headers) takes around 12.5 ms for the round trip at 9600 bits/sec. This is well within the 100–200 ms range mentioned earlier. Also notice that compressing the headers from 40 bytes to 5 bytes reduces the round-trip time for the 1 byte of data from 85 to 12.5 ms.

Unfortunately these types of calculations are harder to make when newer error correcting, compressing modems are being used. The compression employed by these modems reduces the number of bytes sent across the wire, but the error correction may increase the amount of time to transfer these bytes. Nevertheless, these calculations give us a starting point to make reasonable decisions.

In later chapters we'll use these serial line calculations to verify some of the timings that we see when watching packets go across a serial link.

2.11 Summary

This chapter has examined the lowest layer in the Internet protocol suite, the link layer. We looked at the difference between Ethernet and IEEE 802.2/802.3 encapsulation, and the encapsulation used by SLIP and PPP. Since both SLIP and PPP are often used on slow links, both provide a way to compress the common fields that don't often change. This provides better interactive response.

The loopback interface is provided by most implementations. Access to this interface is either through the special loopback address, normally 127.0.0.1, or by sending IP

datagrams to one of the host's own IP addresses. Loopback data has been completely processed by the transport layer and by IP when it loops around to go up the protocol stack.

We described an important feature of many link layers, the MTU, and the related concept of a path MTU. Using the typical MTUs for serial lines, we calculated the latency involved in SLIP and CSLIP links.

This chapter has covered only a few of the common data-link technologies used with TCP/IP today. One reason for the success of TCP/IP is its ability to work on top of almost any data-link technology.

Exercises

2.1 If your system supports the `netstat`(1) command (see Section 3.9 also), use it to determine the interfaces on your system and their MTUs.

3

IP: Internet Protocol

3.1 Introduction

IP is the workhorse protocol of the TCP/IP protocol suite. All TCP, UDP, ICMP, and IGMP data gets transmitted as IP datagrams (Figure 1.4). A fact that amazes many newcomers to TCP/IP, especially those from an X.25 or SNA background, is that IP provides an unreliable, connectionless datagram delivery service.

By *unreliable* we mean there are no guarantees that an IP datagram successfully gets to its destination. IP provides a best effort service. When something goes wrong, such as a router temporarily running out of buffers, IP has a simple error handling algorithm: throw away the datagram and try to send an ICMP message back to the source. Any required reliability must be provided by the upper layers (e.g., TCP).

The term *connectionless* means that IP does not maintain any state information about successive datagrams. Each datagram is handled independently from all other datagrams. This also means that IP datagrams can get delivered out of order. If a source sends two consecutive datagrams (first A, then B) to the same destination, each is routed independently and can take different routes, with B arriving before A.

In this chapter we take a brief look at the fields in the IP header, describe IP routing, and cover subnetting. We also look at two useful commands: `ifconfig` and `netstat`. We leave a detailed discussion of some of the fields in the IP header for later when we can see exactly how the fields are used. RFC 791 [Postel 1981a] is the official specification of IP.

3.2 IP Header

Figure 3.1 shows the format of an IP datagram. The normal size of the IP header is 20 bytes, unless options are present.

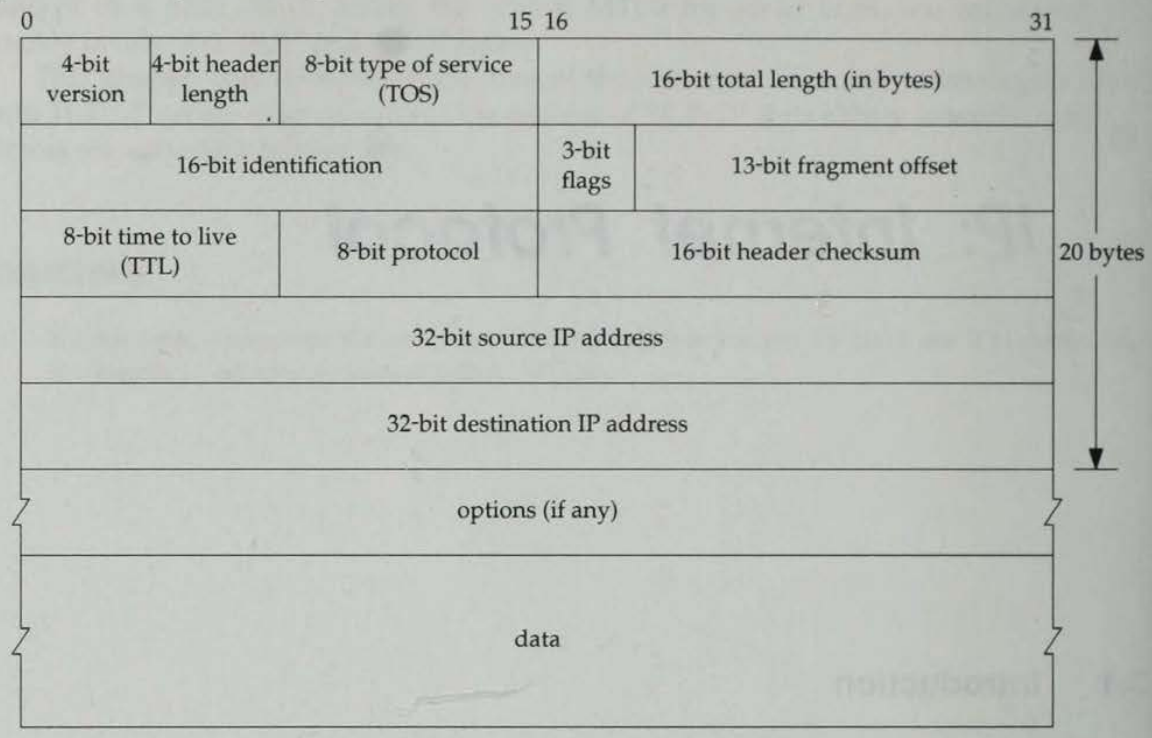


Figure 3.1 IP datagram, showing the fields in the IP header.

We will show the pictures of protocol headers in TCP/IP as in Figure 3.1. The most significant bit is numbered 0 at the left, and the least significant bit of a 32-bit value is numbered 31 on the right.

The 4 bytes in the 32-bit value are transmitted in the order: bits 0–7 first, then bits 8–15, then 16–23, and bits 24–31 last. This is called *big endian* byte ordering, which is the byte ordering required for all binary integers in the TCP/IP headers as they traverse a network. This is called the *network byte order*. Machines that store binary integers in other formats, such as the *little endian* format, must convert the header values into the network byte order before transmitting the data.

The current protocol *version* is 4, so IP is sometimes called IPv4. Section 3.10 discusses some proposals for a new version of IP.

The *header length* is the number of 32-bit words in the header, including any options. Since this is a 4-bit field, it limits the header to 60 bytes. In Chapter 8 we'll see that this limitation makes some of the options, such as the record route option, useless today. The normal value of this field (when no options are present) is 5.

The *type-of-service* field (TOS) is composed of a 3-bit precedence field (which is ignored today), 4 TOS bits, and an unused bit that must be 0. The 4 TOS bits are: minimize delay, maximize throughput, maximize reliability, and minimize monetary cost.

Only 1 of these 4 bits can be turned on. If all 4 bits are 0 it implies normal service. RFC 1340 [Reynolds and Postel 1992] specifies how these bits should be set by all the standard applications. RFC 1349 [Almquist 1992] contains some corrections to this RFC, and a more detailed description of the TOS feature.

Figure 3.2 shows the recommended values of the TOS field for various applications. In the final column we show the hexadecimal value, since that's what we'll see in the `tcpdump` output later in the text.

Application	Minimize delay	Maximize throughput	Maximize reliability	Minimize monetary cost	Hex value
Telnet/Rlogin	1	0	0	0	0x10
FTP					
control	1	0	0	0	0x10
data	0	1	0	0	0x08
any bulk data	0	1	0	0	0x08
TFTP	1	0	0	0	0x10
SMTP					
command phase	1	0	0	0	0x10
data phase	0	1	0	0	0x08
DNS					
UDP query	1	0	0	0	0x10
TCP query	0	0	0	0	0x00
zone transfer	0	1	0	0	0x08
ICMP					
error	0	0	0	0	0x00
query	0	0	0	0	0x00
any IGP	0	0	1	0	0x04
SNMP	0	0	1	0	0x04
BOOTP	0	0	0	0	0x00
NNTP	0	0	0	1	0x02

Figure 3.2 Recommended values for type-of-service field.

The interactive login applications, Telnet and Rlogin, want a minimum delay since they're used interactively by a human for small amounts of data transfer. File transfer by FTP, on the other hand, wants maximum throughput. Maximum reliability is specified for network management (SNMP) and the routing protocols. Usenet news (NNTP) is the only one shown that wants to minimize monetary cost.

The TOS feature is not supported by most TCP/IP implementations today, though newer systems starting with 4.3BSD Reno are setting it. Additionally, new routing protocols such as OSPF and IS–IS are capable of making routing decisions based on this field.

> In Section 2.10 we mentioned that SLIP drivers normally provide type-of-service queueing, allowing interactive traffic to be handled before bulk data. Since most implementations don't use the TOS field, this queueing is done ad hoc by SLIP, with the driver looking at the protocol field (to determine whether it's a TCP segment or not) and then checking the source and destination TCP port numbers to see if the port number corresponds to an interactive service. One driver comments that this "disgusting hack" is required since most implementations don't allow the application to set the TOS field.

The *total length* field is the total length of the IP datagram in bytes. Using this field and the header length field, we know where the data portion of the IP datagram starts, and its length. Since this is a 16-bit field, the maximum size of an IP datagram is 65535 bytes. (Recall from Figure 2.5 [p. 30] that a Hyperchannel has an MTU of 65535. This means there really isn't an MTU—it uses the largest IP datagram possible.) This field also changes when a datagram is fragmented, which we describe in Section 11.5.

Although it's possible to send a 65535-byte IP datagram, most link layers will fragment this. Furthermore, a host is not required to receive a datagram larger than 576 bytes. TCP divides the user's data into pieces, so this limit normally doesn't affect TCP. With UDP we'll encounter numerous applications in later chapters (RIP, TFTP, BOOTP, the DNS, and SNMP) that limit themselves to 512 bytes of user data, to stay below this 576-byte limit. Realistically, however, most implementations today (especially those that support the Network File System, NFS) allow for just over 8192-byte IP datagrams.

The total length field is required in the IP header since some data links (e.g., Ethernet) pad small frames to be a minimum length. Even though the minimum Ethernet frame size is 46 bytes (Figure 2.1), an IP datagram can be smaller. If the total length field wasn't provided, the IP layer wouldn't know how much of a 46-byte Ethernet frame was really an IP datagram.

The *identification* field uniquely identifies each datagram sent by a host. It normally increments by one each time a datagram is sent. We return to this field when we look at fragmentation and reassembly in Section 11.5. Similarly, we'll also look at the *flags* field and the *fragmentation offset* field when we talk about fragmentation.

> RFC 791 [Postel 1981a] says that the identification field should be chosen by the upper layer that is having IP send the datagram. This implies that two consecutive IP datagrams, one generated by TCP and one generated by UDP, can have the same identification field. While this is OK (the reassembly algorithm handles this), most Berkeley-derived implementations have the IP layer increment a kernel variable each time an IP datagram is sent, regardless of which layer passed the data to IP to send. This kernel variable is initialized to a value based on the time-of-day when the system is bootstrapped.

The *time-to-live* field, or *TTL*, sets an upper limit on the number of routers through which a datagram can pass. It limits the lifetime of the datagram. It is initialized by the sender to some value (often 32 or 64) and decremented by one by every router that handles the datagram. When this field reaches 0, the datagram is thrown away, and the sender is notified with an ICMP message. This prevents packets from getting caught in routing loops forever. We return to this field in Chapter 8 when we look at the Traceroute program.

We talked about the *protocol* field in Chapter 1 and showed how it is used by IP to demultiplex incoming datagrams in Figure 1.8. It identifies which protocol gave the data for IP to send.

The *header checksum* is calculated over the IP header only. It does *not* cover any data that follows the header. ICMP, IGMP, UDP, and TCP all have a checksum in their own headers to cover their header and data.

To compute the IP checksum for an outgoing datagram, the value of the checksum field is first set to 0. Then the 16-bit one's complement sum of the header is calculated (i.e., the entire header is considered a sequence of 16-bit words). The 16-bit one's

complement of this sum is stored in the checksum field. When an IP datagram is received, the 16-bit one's complement sum of the header is calculated. Since the receiver's calculated checksum contains the checksum stored by the sender, the receiver's checksum is all one bits if nothing in the header was modified. If the result is not all one bits (a checksum error), IP discards the received datagram. No error message is generated. It is up to the higher layers to somehow detect the missing datagram and retransmit.

ICMP, IGMP, UDP, and TCP all use the same checksum algorithm, although TCP and UDP include various fields from the IP header, in addition to their own header and data. RFC 1071 [Braden, Borman, and Partridge 1988] contains implementation techniques for computing the Internet checksum. Since a router often changes only the TTL field (decrementing it by 1), a router can incrementally update the checksum when it forwards a received datagram, instead of calculating the checksum over the entire IP header again. RFC 1141 [Mallory and Kullberg 1990] describes an efficient way to do this.

> The standard BSD implementation, however, does not use this incremental update feature when forwarding a datagram.

Every IP datagram contains the *source IP address* and the *destination IP address*. These are the 32-bit values that we described in Section 1.4.

The final field, the *options*, is a variable-length list of optional information for the datagram. The options currently defined are:

- security and handling restrictions (for military applications, refer to RFC 1108 [Kent 1991] for details),
- record route (have each router record its IP address, Section 7.3),
- timestamp (have each router record its IP address and time, Section 7.4),
- loose source routing (specifying a list of IP addresses that must be traversed by the datagram, Section 8.5), and
- strict source routing (similar to loose source routing but here only the addresses in the list can be traversed, Section 8.5).

These options are rarely used and not all host and routers support all the options.

The options field always ends on a 32-bit boundary. Pad bytes with a value of 0 are added if necessary. This assures that the IP header is always a multiple of 32 bits (as required for the *header length* field).

3.3 IP Routing

Conceptually, IP routing is simple, especially for a host. If the destination is directly connected to the host (e.g., a point-to-point link) or on a shared network (e.g., Ethernet or token ring), then the IP datagram is sent directly to the destination. Otherwise the

host sends the datagram to a default router, and lets the router deliver the datagram to its destination. This simple scheme handles most host configurations.

In this section and in Chapter 9 we'll look at the more general case where the IP layer can be configured to act as a router in addition to acting as a host. Most multiuser systems today, including almost every Unix system, can be configured to act as a router. We can then specify a single routing algorithm that both hosts and routers can use. The fundamental difference is that a host *never* forwards datagrams from one of its interfaces to another, while a router forwards datagrams. A host that contains embedded router functionality should never forward a datagram unless it has been specifically configured to do so. We say more about this configuration option in Section 9.4.

In our general scheme, IP can receive a datagram from TCP, UDP, ICMP, or IGMP (that is, a locally generated datagram) to send, or one that has been received from a network interface (a datagram to forward). The IP layer has a routing table in memory that it searches each time it receives a datagram to send. When a datagram is received from a network interface, IP first checks if the destination IP address is one of its own IP addresses or an IP broadcast address. If so, the datagram is delivered to the protocol module specified by the protocol field in the IP header. If the datagram is not destined for this IP layer, then (1) if the IP layer was configured to act as a router the packet is forwarded (that is, handled as an outgoing datagram as described below), else (2) the datagram is silently discarded.

Each entry in the routing table contains the following information:

- Destination IP address. This can be either a complete *host address* or a *network address*, as specified by the flag field (described below) for this entry. A host address has a nonzero host ID (Figure 1.5) and identifies one particular host, while a network address has a host ID of 0 and identifies all the hosts on that network (e.g., Ethernet, token ring).
- IP address of a *next-hop router*, or the IP address of a directly connected network. A next-hop router is one that is on a directly connected network to which we can send datagrams for delivery. The next-hop router is not the final destination, but it takes the datagrams we send it and forwards them to the final destination.
- Flags. One flag specifies whether the destination IP address is the address of a network or the address of a host. Another flag says whether the next-hop router field is really a next-hop router or a directly connected interface. (We describe each of these flags in Section 9.2.)
- Specification of which network interface the datagram should be passed to for transmission.

IP routing is done on a hop-by-hop basis. As we can see from this routing table information, IP does not know the complete route to any destination (except, of course, those destinations that are directly connected to the sending host). All that IP routing provides is the IP address of the next-hop router to which the datagram is sent. It is assumed that the next-hop router is really "closer" to the destination than the sending host is, and that the next-hop router is directly connected to the sending host.

IP routing performs the following actions:

1. Search the routing table for an entry that matches the complete destination IP address (matching network ID and host ID). If found, send the packet to the indicated next-hop router or to the directly connected interface (depending on the flags field). Point-to-point links are found here, for example, since the other end of such a link is the other host's complete IP address.
2. Search the routing table for an entry that matches just the destination network ID. If found, send the packet to the indicated next-hop router or to the directly connected interface (depending on the flags field). All the hosts on the destination network can be handled with this single routing table entry. All the hosts on a local Ethernet, for example, are handled with a routing table entry of this type.

   This check for a network match must take into account a possible subnet mask, which we describe in the next section.
3. Search the routing table for an entry labeled "default." If found, send the packet to the indicated next-hop router.

If none of the steps works, the datagram is undeliverable. If the undeliverable datagram was generated on this host, a "host unreachable" or "network unreachable" error is normally returned to the application that generated the datagram.

A complete matching host address is searched for before a matching network ID. Only if both of these fail is a default route used. Default routes, along with the ICMP redirect message sent by a next-hop router (if we chose the wrong default for a datagram), are powerful features of IP routing that we'll come back to in Chapter 9.

The ability to specify a route to a network, and not have to specify a route to every host, is another fundamental feature of IP routing. Doing this allows the routers on the Internet, for example, to have a routing table with thousands of entries, instead of a routing table with more than one million entries.

Examples

First consider a simple example: our host `bsdi` has an IP datagram to send to our host `sun`. Both hosts are on the same Ethernet (see inside front cover). Figure 3.3 shows the delivery of the datagram.

When IP receives the datagram from one of the upper layers it searches its routing table and finds that the destination IP address (140.252.13.33) is on a directly connected network (the Ethernet 140.252.13.0). A matching network address is found in the routing table. (In the next section we'll see that because of subnetting the network address of this Ethernet is really 140.252.13.32, but that doesn't affect this discussion of routing.)

The datagram is passed to the Ethernet device driver, and sent to `sun` as an Ethernet frame (Figure 2.1). The destination address in the IP datagram is Sun's IP address (140.252.13.33) and the destination address in the link-layer header is the 48-bit Ethernet address of `sun`'s Ethernet interface. This 48-bit Ethernet address is obtained using ARP, as we describe in the next chapter.

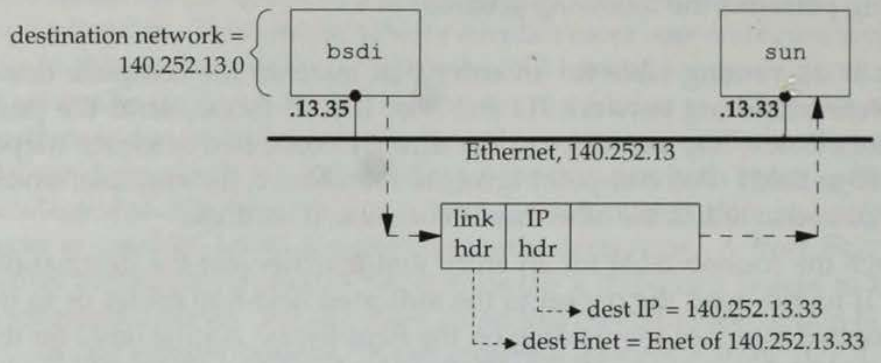


Figure 3.3 Delivery of IP datagram from `bsdi` to `sun`.

Now consider another example: `bsdi` has an IP datagram to send to the host `ftp.uu.net`, whose IP address is 192.48.96.9. Figure 3.4 shows the path of the datagram through the first three routers. First `bsdi` searches its routing table but doesn't find a matching host entry or a matching network entry. It uses its default entry, which tells it to send datagrams to `sun`, the next-hop router. When the datagram travels from `bsdi` to `sun` the destination IP address is the final destination (192.48.96.9) but the link-layer address is the 48-bit Ethernet address of `sun`'s Ethernet interface. Compare this datagram with the one in Figure 3.3, where the destination IP address and the destination link-layer address specified the same host (`sun`).

When `sun` receives the datagram it realizes that the datagram's destination IP address is not one of its own, and `sun` is configured to act as a router, so it forwards the datagram. Its routing table is searched and the default entry is used. The default entry on `sun` tells it to send datagrams to the next-hop router `netb`, whose IP address is 140.252.1.183. The datagram is sent across the point-to-point SLIP link, using the minimal encapsulation we showed in Figure 2.2. We don't show a link-layer header, as we do on the Ethernets, because there isn't one on a SLIP link.

When `netb` receives the datagram it goes through the same steps that `sun` just did: the datagram is not destined for one of its own IP addresses, and `netb` is configured to act as a router, so the datagram is forwarded. The default routing table entry is used, sending the datagram to the next-hop router `gateway` (140.252.1.4). ARP is used by `netb` on the Ethernet 140.252.1 to obtain the 48-bit Ethernet address corresponding to 140.252.1.4, and that Ethernet address is the destination address in the link-layer header.

`gateway` goes through the same steps as the previous two routers and its default routing table entry specifies 140.252.104.2 as the next-hop router. (We'll verify that this is the next-hop router for `gateway` using Traceroute in Figure 8.4.)

A few key points come out in this example.

1. All the hosts and routers in this example used a default route. Indeed, most hosts and some routers can use a default route for everything other than destinations on local networks.

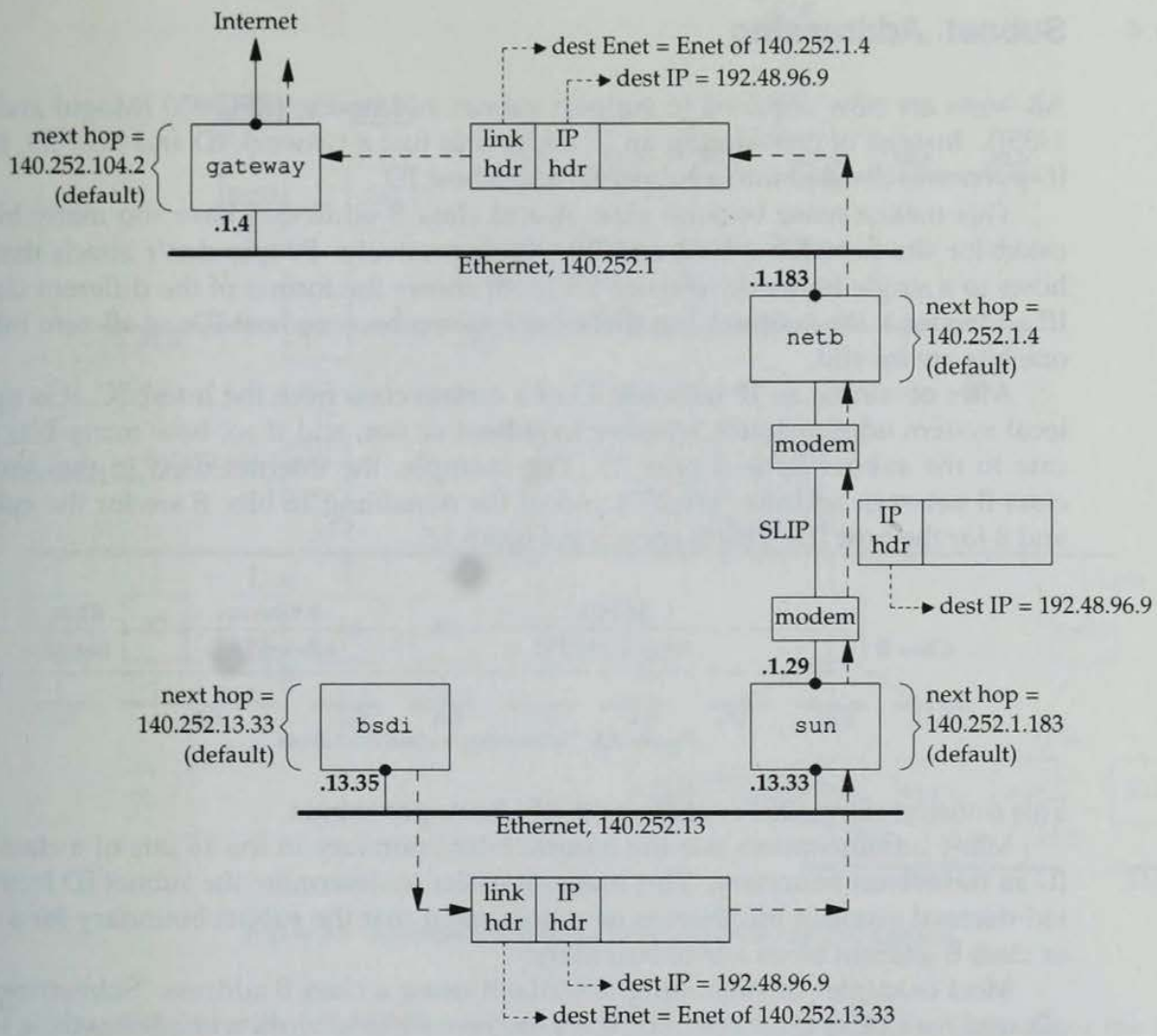


Figure 3.4 Initial path of datagram from `bsdi` to `ftp.uu.net` (192.48.96.9).

2. The destination IP address in the datagram never changes. (In Section 8.5 we'll see that this is not true only if source routing is used, which is rare.) All the routing decisions are based on this destination address.
3. A different link-layer header can be used on each link, and the link-layer destination address (if present) always contains the link-layer address of the next hop. In our example both Ethernets encapsulated a link-layer header containing the next-hop's Ethernet address, but the SLIP link did not. The Ethernet addresses are normally obtained using ARP.

In Chapter 9 we'll look at IP routing again, after describing ICMP. We'll also look at some sample routing tables and how they're used for routing decisions.

3.4 Subnet Addressing

All hosts are now required to support subnet addressing (RFC 950 [Mogul and Postel 1985]). Instead of considering an IP address as just a network ID and host ID, the host ID portion is divided into a subnet ID and a host ID.

This makes sense because class A and class B addresses have too many bits allocated for the host ID: $2^{24} - 2$ and $2^{16} - 2$, respectively. People don't attach that many hosts to a single network. (Figure 1.5 [p. 8] shows the format of the different classes of IP addresses.) We subtract 2 in these expressions because host IDs of all zero bits or all one bits are invalid.

After obtaining an IP network ID of a certain class from the InterNIC, it is up to the local system administrator whether to subnet or not, and if so, how many bits to allocate to the subnet ID and host ID. For example, the internet used in this text has a class B network address (140.252) and of the remaining 16 bits, 8 are for the subnet ID and 8 for the host ID. This is shown in Figure 3.5.

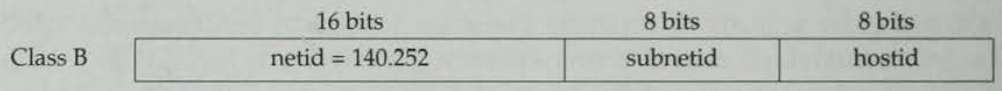


Figure 3.5 Subnetting a class B address.

This division allows 254 subnets, with 254 hosts per subnet.

Many administrators use the natural 8-bit boundary in the 16 bits of a class B host ID as the subnet boundary. This makes it easier to determine the subnet ID from a dotted-decimal number, but there is no requirement that the subnet boundary for a class A or class B address be on a byte boundary.

Most examples of subnetting describe it using a class B address. Subnetting is also allowed for a class C address, but there are fewer bits to work with. Subnetting is rarely shown with a class A address because there are so few class A addresses. (Most class A addresses are, however, subnetted.)

Subnetting hides the details of internal network organization (within a company or campus) to external routers. Using our example network, all IP addresses have the class B network ID of 140.252. But there are more than 30 subnets and more than 400 hosts distributed over those subnets. A single router provides the connection to the Internet, as shown in Figure 3.6.

In this figure we have labeled most of the routers as R*n*, where *n* is the subnet number. We show the routers that connect these subnets, along with the nine systems from the figure on the inside front cover. The Ethernets are shown as thicker lines, and the point-to-point links as dashed lines. We do *not* show all the hosts on the various subnets. For example, there are more than 50 hosts on the 140.252.3 subnet, and more than 100 on the 140.252.1 subnet.

The advantage to using a single class B address with 30 subnets, compared to 30 class C addresses, is that subnetting reduces the size of the Internet's routing tables. The fact that the class B address 140.252 is subnetted is transparent to all Internet routers other than the ones within the 140.252 subnet. To reach any host whose IP

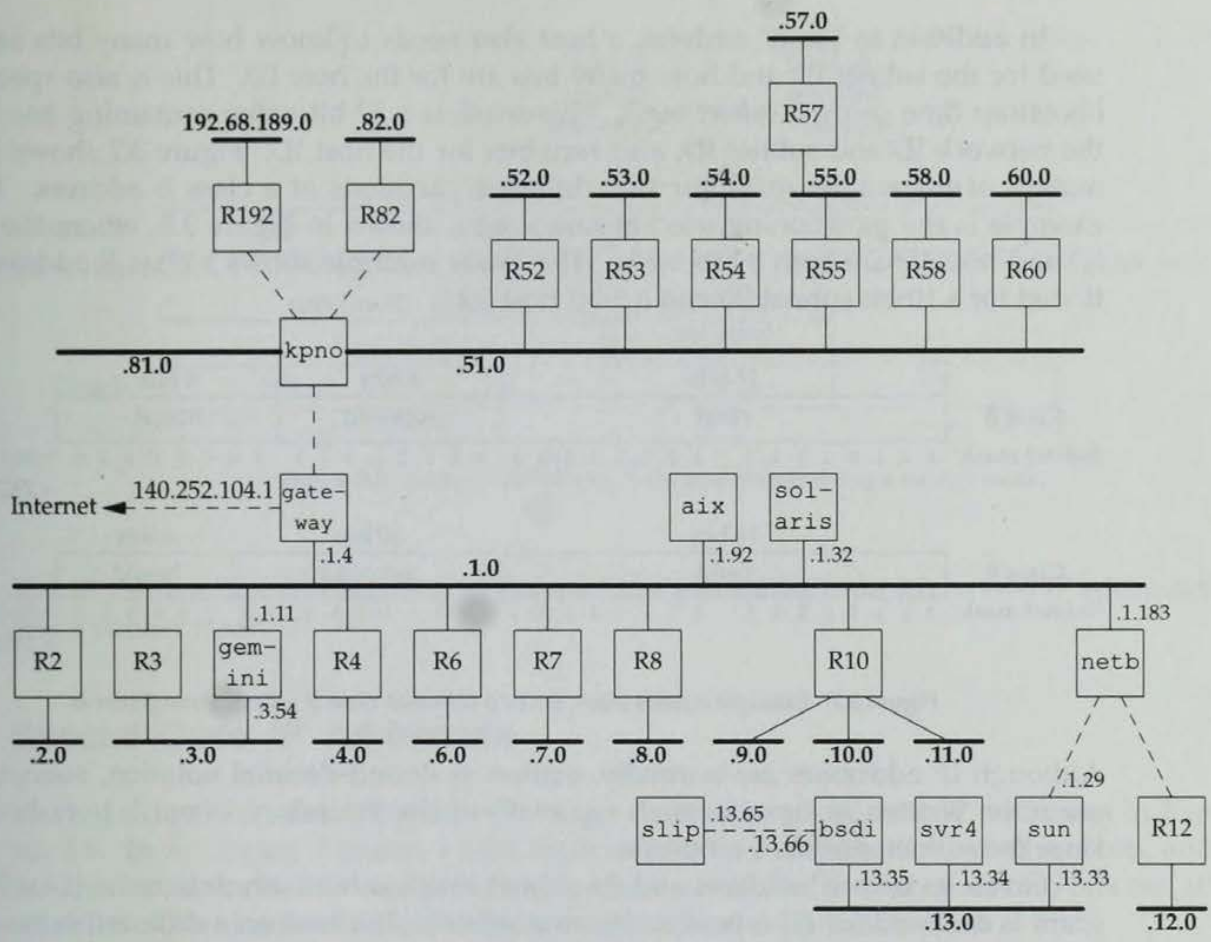


Figure 3.6 Arrangement of most of the noao.edu 140.252 subnets.

address begins with 140.252, the external routers only need to know the path to the IP address 140.252.104.1. This means that only one routing table entry is needed for all the 140.252 networks, instead of 30 entries if 30 class C addresses were used. Subnetting, therefore, reduces the size of routing tables. (In Section 10.8 we'll look at a new technique that helps reduce the size of routing tables even if class C addresses are used.)

To show that subnetting is not transparent to routers within the subnet, assume in Figure 3.6 that a datagram arrives at `gateway` from the Internet with a destination address of 140.252.57.1. The router `gateway` needs to know that the subnet number is 57, and that datagrams for this subnet are sent to `kpno`. Similarly `kpno` must send the datagram to R55, who then sends it to R57.

3.5 Subnet Mask

Part of the configuration of any host that takes place at bootstrap time is the specification of the host's IP address. Most systems have this stored in a disk file that's read at bootstrap time, and we'll see in Chapter 5 how a diskless system can also find out its IP address when it's bootstrapped.

In addition to the IP address, a host also needs to know how many bits are to be used for the subnet ID and how many bits are for the host ID. This is also specified at bootstrap time using a *subnet mask*. This mask is a 32-bit value containing one bits for the network ID and subnet ID, and zero bits for the host ID. Figure 3.7 shows the formation of the subnet mask for two different partitions of a class B address. The top example is the partitioning used at `noao.edu`, shown in Figure 3.5, where the subnet ID and host ID are both 8 bits wide. The lower example shows a class B address partitioned for a 10-bit subnet ID and a 6-bit host ID.

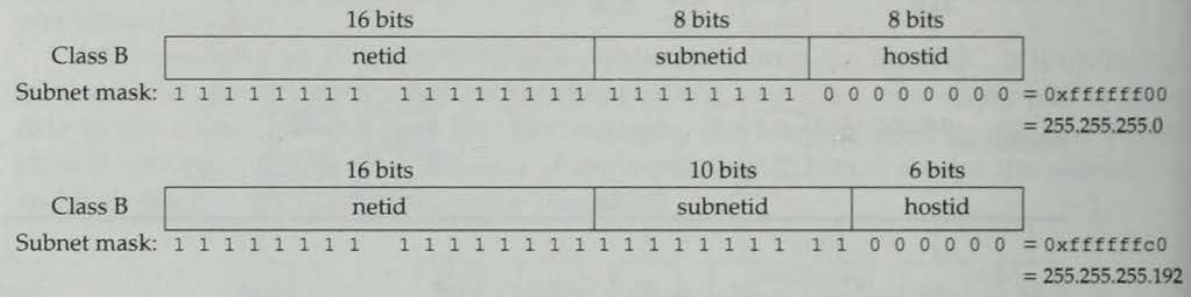


Figure 3.7 Example subnet mask for two different class B subnet arrangements.

Although IP addresses are normally written in dotted-decimal notation, subnet masks are often written in hexadecimal, especially if the boundary is not a byte boundary, since the subnet mask is a bit mask.

Given its own IP address and its subnet mask, a host can determine if an IP datagram is destined for (1) a host on its own subnet, (2) a host on a different subnet on its own network, or (3) a host on a different network. Knowing your own IP address tells you whether you have a class A, B, or C address (from the high-order bits), which tells you where the boundary is between the network ID and the subnet ID. The subnet mask then tells you where the boundary is between the subnet ID and the host ID.

Example

Assume our host address is 140.252.1.1 (a class B address) and our subnet mask is 255.255.255.0 (8 bits for the subnet ID and 8 bits for the host ID).

- If a destination IP address is 140.252.4.5, we know that the class B network IDs are the same (140.252), but the subnet IDs are different (1 and 4). Figure 3.8 shows how this comparison of two IP addresses is done, using the subnet mask.
- If the destination IP address is 140.252.1.22, the class B network IDs are the same (140.252), and the subnet IDs are the same (1). The host IDs, however, are different.
- If the destination IP address is 192.43.235.6 (a class C address), the network IDs are different. No further comparisons can be made against this address.

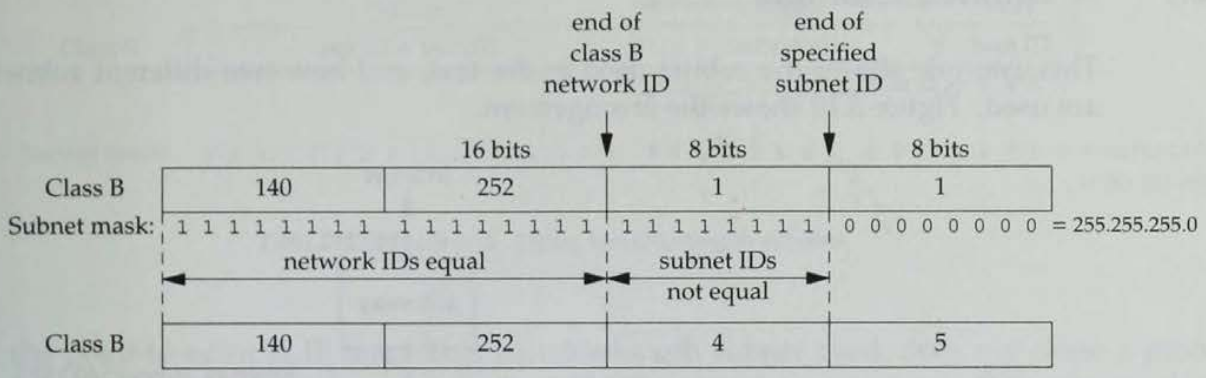


Figure 3.8 Comparison of two class B addresses using a subnet mask.

The IP routing function makes comparisons like this all the time, given two IP addresses and a subnet mask.

3.6 Special Case IP Addresses

Having described subnetting we now show the seven special case IP addresses in Figure 3.9. In this figure, 0 means a field of all zero bits, –1 means a field of all one bits, and *netid*, *subnetid*, and *hostid* mean the corresponding field that is neither all zero bits nor all one bits. A blank subnet ID column means the address is not subnetted.

IP address			Can appear as		Description
net ID	subnet ID	host ID	source?	destination?	
0		0	OK	never	this host on this net (see restrictions below)
0		*hostid*	OK	never	specified host on this net (see restrictions below)
127		*anything*	OK	OK	loopback address (Section 2.7)
–1		–1	never	OK	limited broadcast (never forwarded)
netid		–1	never	OK	net-directed broadcast to *netid*
netid	*subnetid*	–1	never	OK	subnet-directed broadcast to *netid*, *subnetid*
netid	–1	–1	never	OK	all-subnets-directed broadcast to *netid*

Figure 3.9 Special case IP addresses.

We have divided this table into three sections. The first two entries are special case source addresses, the next one is the special loopback address, and the final four are the broadcast addresses.

The first two entries in the table, with a network ID of 0, can only appear as the source address as part of an initialization procedure when a host is determining its own IP address, for example, when the BOOTP protocol is being used (Chapter 16).

In Section 12.2 we'll examine the four types of broadcast addresses in more detail.

3.7 A Subnet Example

This example shows the subnet used in the text, and how two different subnet masks are used. Figure 3.10 shows the arrangement.

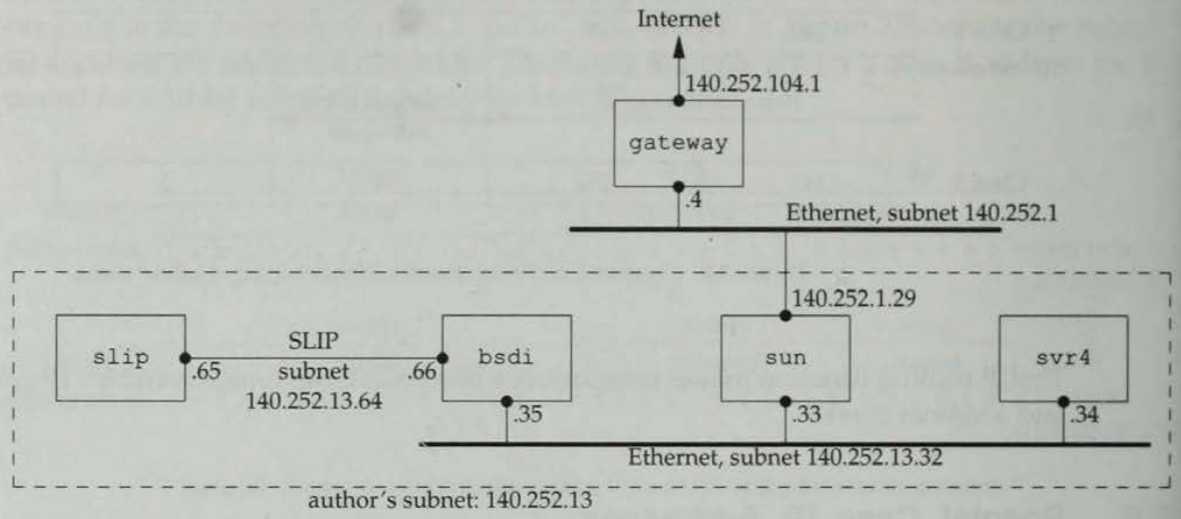


Figure 3.10 Arrangement of hosts and networks for author's subnet.

If you compare this figure with the one on the inside front cover, you'll notice that we've omitted the detail that the connection from the router sun to the top Ethernet in Figure 3.10 is really a dialup SLIP connection. This detail doesn't affect our description of subnetting in this section. We'll return to this detail in Section 4.6 when we describe proxy ARP.

The problem is that we have two separate networks within subnet 13: an Ethernet and a point-to-point link (the hardwired SLIP link). (Point-to-point links always cause problems since each end normally requires an IP address.) There could be more hosts and networks in the future, but not enough hosts across the different networks to justify using another subnet number. Our solution is to extend the subnet ID from 8 to 11 bits, and decrease the host ID from 8 to 5 bits. This is called *variable-length subnets* since most networks within the 140.252 network use an 8-bit subnet mask while our network uses an 11-bit subnet mask.

> RFC 1009 [Braden and Postel 1987] allows a subnetted network to use more than one subnet mask. The new Router Requirements RFC [Almquist 1993] requires support for this.
>
> The problem, however, is that not all routing protocols exchange the subnet mask along with the destination network ID. We'll see in Chapter 10 that RIP does not support variable-length subnets, while RIP Version 2 and OSPF do. We don't have a problem with our example, since RIP isn't required on the author's subnet.

Figure 3.11 shows the IP address structure used within the author's subnet. The first 8 bits of the 11-bit subnet ID are always 13 within the author's subnet. For the remaining 3 bits of the subnet ID, we use binary 001 for the Ethernet, and binary 010 for

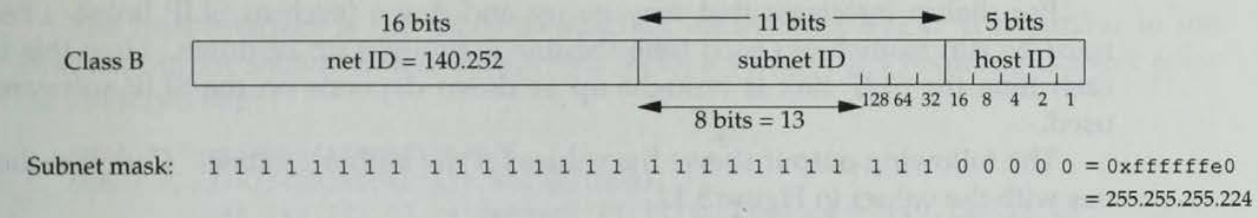


Figure 3.11 Using variable-length subnets.

the point-to-point SLIP link. This variable-length subnet mask does not cause a problem for other hosts and routers in the 140.252 network—as long as all datagrams destined for the subnet 140.252.13 are sent to the router `sun` (IP address 140.252.1.29) in Figure 3.10, and if `sun` knows about the 11-bit subnet ID for the hosts on its subnet 13, everything is fine.

The subnet mask for all the interfaces on the 140.252.13 subnet is 255.255.255.224, or `0xffffffe0`. This indicates that the rightmost 5 bits are for the host ID, and the 27 bits to the left are the network ID and subnet ID.

Figure 3.12 shows the allocation of IP addresses and subnet masks for the interfaces shown in Figure 3.10.

Host	IP address	Subnet mask	Net ID/Subnet ID	Host ID	Comment
`sun`	140.252.1.29	255.255.255.0	140.252.1	29	on subnet 1
	140.252.13.33	255.255.255.224	140.252.13.32	1	on author's Ethernet
`svr4`	140.252.13.34	255.255.255.224	140.252.13.32	2	
`bsdi`	140.252.13.35	255.255.255.224	140.252.13.32	3	on Ethernet
	140.252.13.66	255.255.255.224	140.252.13.64	2	point-to-point
`slip`	140.252.13.65	255.255.255.224	140.252.13.64	1	point-to-point
	140.252.13.63	255.255.255.224	140.252.13.32	31	broadcast addr on Ethernet

Figure 3.12 IP addresses on author's subnet.

The first column is labeled "Host," but both `sun` and `bsdi` also act as routers, since they are multihomed and route packets from one interface to another.

The final row in this table notes that the broadcast address for the Ethernet in Figure 3.10 is 140.252.13.63: it is formed from the subnet ID of the Ethernet (140.252.13.32) and the low-order 5 bits in Figure 3.11 set to 1 (16+8+4+2+1 = 31). (We'll see in Chapter 12 that this address is called the subnet-directed broadcast address.)

3.8 `ifconfig` Command

Now that we've described the link layer and the IP layer we can show the command used to configure or query a network interface for use by TCP/IP. The `ifconfig`(8) command is normally run at bootstrap time to configure each interface on a host.

For dialup interfaces that may go up and down (such as SLIP links), `ifconfig` must be run (somehow) each time the line is brought up or down. How this is done each time the SLIP link is brought up or down depends on the SLIP software being used.

The following output shows the values for the author's subnet. Compare these values with the values in Figure 3.12.

```
sun % /usr/etc/ifconfig -a          SunOS -a option says report on all interfaces
le0: flags=63<UP,BROADCAST,NOTRAILERS,RUNNING>
        inet 140.252.13.33 netmask ffffffe0 broadcast 140.252.13.63
sl0: flags=1051<UP,POINTOPOINT,RUNNING,LINK0>
        inet 140.252.1.29 --> 140.252.1.183 netmask ffffff00
lo0: flags=49<UP,LOOPBACK,RUNNING>
        inet 127.0.0.1 netmask ff000000
```

The loopback interface (Section 2.7) is considered a network interface. Its class A address is not subnetted.

Other things to notice are that trailer encapsulation (Section 2.3) is not used on the Ethernet, and that the Ethernet is capable of broadcasting, while the SLIP link is a point-to-point link.

The flag `LINK0` for the SLIP interface is the configuration option that enables compressed slip (CSLIP, Section 2.5). Other possible options are `LINK1`, which enables CSLIP if a compressed packet is received from the other end, and `LINK2`, which causes all outgoing ICMP packets to be thrown away. We'll look at the destination address of this SLIP link in Section 4.6.

> A comment in the installation instructions gives the reason for this last option: "This shouldn't have to be set, but some cretin pinging you can drive your throughput to zero."

`bsdi` is the other router. Since the `-a` option is a SunOS feature, we have to execute `ifconfig` multiple times, specifying the interface name as an argument:

```
bsdi % /sbin/ifconfig we0
we0: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,SIMPLEX>
        inet 140.252.13.35 netmask ffffffe0 broadcast 140.252.13.63
bsdi % /sbin/ifconfig sl0
sl0: flags=1011<UP,POINTOPOINT,LINK0>
        inet 140.252.13.66 --> 140.252.13.65 netmask ffffffe0
```

Here we see a new option for the Ethernet interface (`we0`): `SIMPLEX`. This 4.4BSD flag specifies that the interface can't hear its own transmissions. It is set in BSD/386 for all the Ethernet interfaces. When set, if the interface is sending a frame to the broadcast address, a copy is made for the local host and sent to the loopback address. (We show an example of this feature in Section 6.3.)

On the host `slip` the configuration of the SLIP interface is nearly identical to the output shown above on `bsdi`, with the exception that the IP addresses of the two ends are swapped:

```
slip % /sbin/ifconfig sl0
sl0: flags=1011<UP,POINTOPOINT,LINK0>
        inet 140.252.13.65 --> 140.252.13.66 netmask ffffffe0
```

The final interface is the Ethernet interface on the host `svr4`. It is similar to the Ethernet output shown earlier, except that SVR4's version of `ifconfig` doesn't print the `RUNNING` flag:

```
svr4 % /usr/sbin/ifconfig emd0
emd0: flags=23<UP,BROADCAST,NOTRAILERS>
        inet 140.252.13.34 netmask fffffe0 broadcast 140.252.13.63
```

The `ifconfig` command normally supports other protocol families (other than TCP/IP) and has numerous additional options. Check your system's manual for these details.

3.9 `netstat` Command

The `netstat`(1) command also provides information about the interfaces on a system. The `-i` flag prints the interface information, and the `-n` flag prints IP addresses instead of hostnames.

```
sun % netstat -in
Name  Mtu  Net/Dest      Address        Ipkts  Ierrs Opkts  Oerrs Collis Queue
le0   1500 140.252.13.32 140.252.13.33  67719  0     92133  0     1      0
sl0   552  140.252.1.183 140.252.1.29   48035  0     54963  0     0      0
lo0   1536 127.0.0.0     127.0.0.1      15548  0     15548  0     0      0
```

This command prints the MTU of each interface, the number of input packets, input errors, output packets, output errors, collisions, and the current size of the output queue.

We'll return to the `netstat` command in Chapter 9 when we use it to examine the routing table, and in Chapter 13 when we use a modified version to see active multicast groups.

3.10 IP Futures

There are three problems with IP. They are a result of the phenomenal growth of the Internet over the past few years. (See Exercise 1.2 also.)

1. Over half of all class B addresses have already been allocated. Current estimates predict exhaustion of the class B address space around 1995, if they continue to be allocated as they have been in the past.
2. 32-bit IP addresses in general are inadequate for the predicted long-term growth of the Internet.
3. The current routing structure is not hierarchical, but flat, requiring one routing table entry per network. As the number of networks grows, amplified by the allocation of multiple class C addresses to a site with multiple networks, instead of a single class B address, the size of the routing tables grows.

CIDR (Classless Interdomain Routing) proposes a fix to the third problem that will extend the usefulness of the current version of IP (IP version 4) into the next century. We discuss it in more detail in Section 10.8.

Four proposals have been made for a new version of IP, often called *IPng*, for the next generation of IP. The May 1993 issue of *IEEE Network* (vol. 7, no. 3) contains overviews of the first three proposals, along with an article on CIDR. RFC 1454 [Dixon 1993] also compares the first three proposals.

1. SIP, the Simple Internet Protocol. It proposes a minimal set of changes to IP that uses 64-bit addresses and a different header format. (The first 4 bits of the header still contain the version number, with a value other than 4.)
2. PIP. This proposal also uses larger, variable-length, hierarchical addresses with a different header format.
3. TUBA, which stands for "TCP and UDP with Bigger Addresses," is based on the OSI CLNP (Connectionless Network Protocol), an OSI protocol similar to IP. It provides much larger addresses: variable length, up to 20 bytes. Since CLNP is an existing protocol, whereas SIP and PIP are just proposals, documentation already exists on CLNP. RFC 1347 [Callon 1992] provides details on TUBA. Chapter 7 of [Perlman 1992] contains a comparison of IPv4 and CLNP. Many routers already support CLNP, but few hosts do.
4. TP/IX, which is described in RFC 1475 [Ullmann 1993]. As with SIP, it uses 64 bits for IP addresses, but it also changes the TCP and UDP headers: 32-bit port number for both protocols, along with 64-bit sequence numbers, 64-bit acknowledgment numbers, and 32-bit windows for TCP.

The first three proposals use basically the same versions of TCP and UDP as the transport layers.

Since only one of these four proposals will be chosen as the successor to IPv4, and since the decision may have been made by the time you read this, we won't say any more about them. With the forthcoming implementation of CIDR to handle the short-term problem, it will take many years to implement the successor to IPv4.

3.11 Summary

We started this chapter with a description of the IP header and briefly described all the fields in this header. We also gave an introduction to IP routing, and saw that host routing can be simple: the destination is either on a directly connected network, in which case the datagram is sent directly to the destination, or a default router is chosen.

Hosts and routers have a routing table that is used for all routing decisions. There are three types of routes in the table: host specific, network specific, and optional default routes. There is a priority to the entries in a routing table. A host route will be chosen over a network router, and a default route is used only when no other route exists to the destination.

IP routing is done on a hop-by-hop basis. The destination IP address never changes as the datagram proceeds through all the hops, but the encapsulation and destination link-layer address can change on each hop. Most hosts and many routers use a default next-hop router for all nonlocal traffic.

Class A and B addresses are normally subnetted. The number of bits used for the subnet ID is specified by the subnet mask. We gave a detailed example of this, using the author's subnet, and introduced variable-length subnets. The use of subnetting reduces the size of the Internet routing tables, since many networks can often be accessed through a single point. Information on the interfaces and networks is available through the `ifconfig` and `netstat` commands. This includes the IP address of the interface, its subnet mask, broadcast address, and MTU.

We finished the chapter with a discussion of potential changes to the Internet protocol suite–the next generation of IP.

Exercises

3.1 Must the loopback address be 127.0.0.1?

3.2 Identify the routers in Figure 3.6 with more than two network interfaces.

3.3 What's the difference in the subnet mask for a class A address with 16 bits for the subnet ID and a class B address with 8 bits for the subnet ID?

3.4 Read RFC 1219 [Tsuchiya 1991] for a recommended technique for assigning subnet IDs and host IDs.

3.5 Is the subnet mask 255.255.0.255 valid for a class A address?

3.6 Why do you think the MTU of the loopback interface printed in Section 3.9 is set to 1536?

3.7 The TCP/IP protocol suite is built on a datagram network technology, the IP layer. Other protocol suites are built on a connection-oriented network technology. Read [Clark 1988] to discover the three advantages the datagram network layer provides.

4

ARP: Address Resolution Protocol

4.1 Introduction

The problem that we deal with in this chapter is that IP addresses only make sense to the TCP/IP protocol suite. A data link such as an Ethernet or a token ring has its own addressing scheme (often 48-bit addresses) to which any network layer using the data link must conform. A network such as an Ethernet can be used by different network layers at the same time. For example, a collection of hosts using TCP/IP and another collection of hosts using some PC network software can share the same physical cable.

When an Ethernet frame is sent from one host on a LAN to another, it is the 48-bit Ethernet address that determines for which interface the frame is destined. The device driver software never looks at the destination IP address in the IP datagram.

Address resolution provides a mapping between the two different forms of addresses: 32-bit IP addresses and whatever type of address the data link uses. RFC 826 [Plummer 1982] is the specification of ARP.

Figure 4.1 shows the two protocols we talk about in this chapter and the next: ARP (address resolution protocol) and RARP (reverse address resolution protocol).

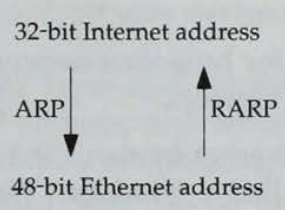


Figure 4.1 Address resolution protocols: ARP and RARP.

ARP provides a dynamic mapping from an IP address to the corresponding hardware address. We use the term *dynamic* since it happens automatically and is normally not a concern of either the application user or the system administrator.

RARP is used by systems without a disk drive (normally diskless workstations or X terminals) but requires manual configuration by the system administrator. We describe it in Chapter 5.

4.2 An Example

Whenever we type a command of the form

```
% ftp bsdi
```

the following steps take place. These numbered steps are shown in Figure 4.2.

1. The application, the FTP client, calls the function `gethostbyname`(3) to convert the hostname (`bsdi`) into its 32-bit IP address. This function is called a *resolver* in the DNS (Domain Name System), which we describe in Chapter 14. This conversion is done using the DNS, or on smaller networks, a static hosts file (`/etc/hosts`).

2. The FTP client asks its TCP to establish a connection with that IP address.

3. TCP sends a connection request segment to the remote host by sending an IP datagram to its IP address. (We'll see the details of how this is done in Chapter 18.)

4. If the destination host is on a locally attached network (e.g., Ethernet, token ring, or the other end of a point-to-point link), the IP datagram can be sent directly to that host. If the destination host is on a remote network, the IP routing function determines the Internet address of a locally attached next-hop router to send the IP datagram to. In either case the IP datagram is sent to a host or router on a locally attached network.

5. Assuming an Ethernet, the sending host must convert the 32-bit IP address into a 48-bit Ethernet address. A translation is required from the *logical* Internet address to its corresponding *physical* hardware address. This is the function of ARP.

   ARP is intended for broadcast networks where many hosts or routers are connected to a single network.

6. ARP sends an Ethernet frame called an *ARP request* to every host on the network. This is called a *broadcast*. We show the broadcast in Figure 4.2 with dashed lines. The ARP request contains the IP address of the destination host (whose name is `bsdi`) and is the request "if you are the owner of this IP address, please respond to me with your hardware address."

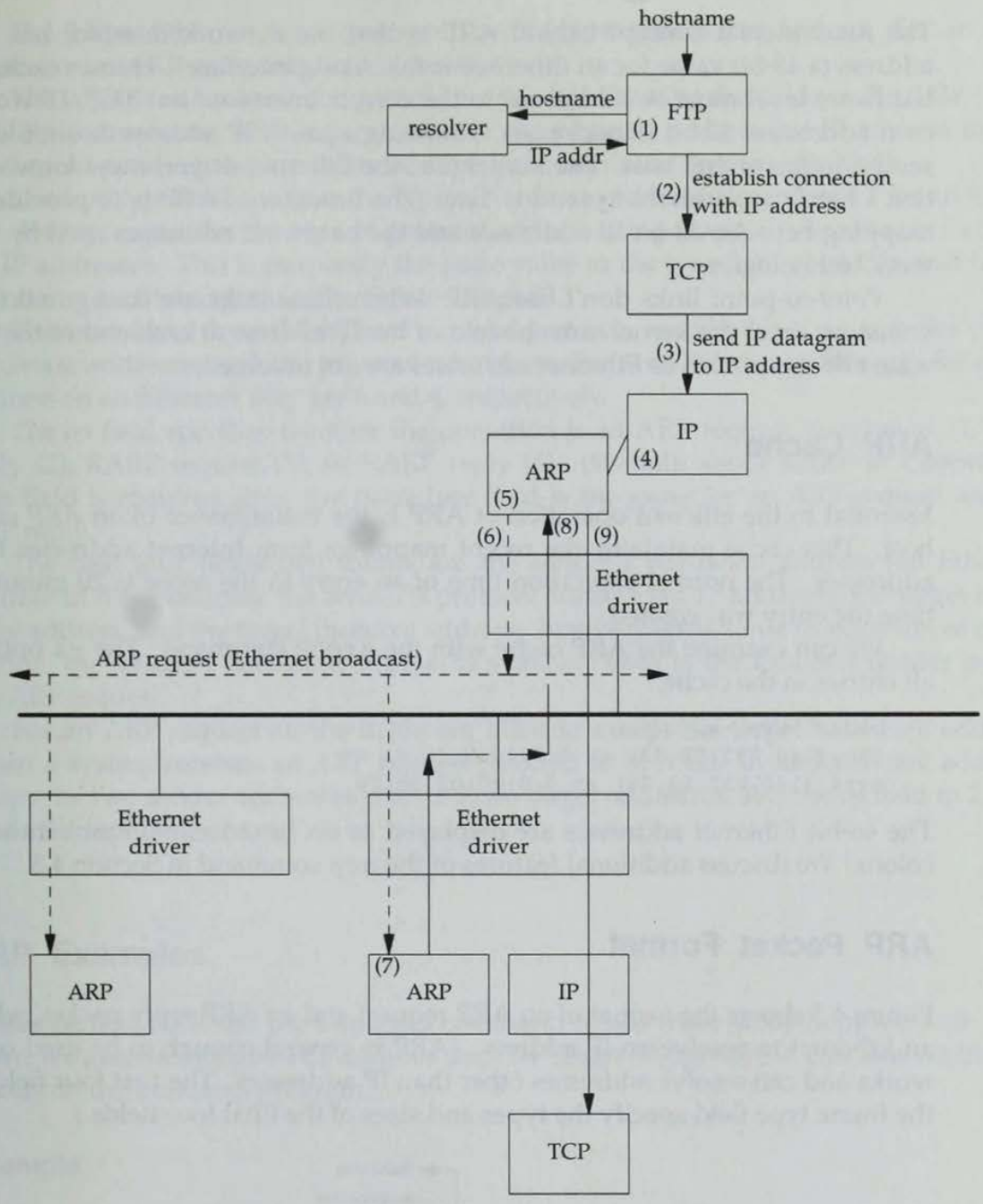


Figure 4.2 Operation of ARP when user types "`ftp` hostname".

7. The destination host's ARP layer receives this broadcast, recognizes that the sender is asking for its IP address, and replies with an *ARP reply*. This reply contains the IP address and the corresponding hardware address.
8. The ARP reply is received and the IP datagram that forced the ARP request–reply to be exchanged can now be sent.
9. The IP datagram is sent to the destination host.

The fundamental concept behind ARP is that the network interface has a hardware address (a 48-bit value for an Ethernet or token ring interface). Frames exchanged at the hardware level must be addressed to the correct interface. But TCP/IP works with its own addresses: 32-bit IP addresses. Knowing a host's IP address doesn't let the kernel send a frame to that host. The kernel (i.e., the Ethernet driver) must know the destination's hardware address to send it data. The function of ARP is to provide a dynamic mapping between 32-bit IP addresses and the hardware addresses used by various network technologies.

Point-to-point links don't use ARP. When these links are configured (normally at bootstrap time) the kernel must be told of the IP address at each end of the link. Hardware addresses such as Ethernet addresses are not involved.

4.3 ARP Cache

Essential to the efficient operation of ARP is the maintenance of an *ARP cache* on each host. This cache maintains the recent mappings from Internet addresses to hardware addresses. The normal expiration time of an entry in the cache is 20 minutes from the time the entry was created.

We can examine the ARP cache with the `arp`(8) command. The `-a` option displays all entries in the cache:

```
bsdi % arp -a
sun (140.252.13.33) at 8:0:20:3:f6:42
svr4 (140.252.13.34) at 0:0:c0:c2:9b:26
```

The 48-bit Ethernet addresses are displayed as six hexadecimal numbers separated by colons. We discuss additional features of the `arp` command in Section 4.8.

4.4 ARP Packet Format

Figure 4.3 shows the format of an ARP request and an ARP reply packet, when used on an Ethernet to resolve an IP address. (ARP is general enough to be used on other networks and can resolve addresses other than IP addresses. The first four fields following the frame type field specify the types and sizes of the final four fields.)

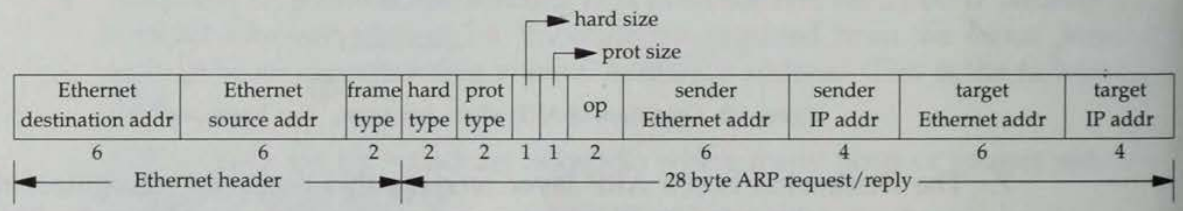


Figure 4.3 Format of ARP request or reply packet when used on an Ethernet.

The first two fields in the Ethernet header are the source and destination Ethernet addresses. The special Ethernet destination address of all one bits means the broadcast address. All Ethernet interfaces on the cable receive these frames.

The 2-byte Ethernet *frame type* specifies the type of data that follows. For an ARP request or an ARP reply, this field is `0x0806`.

The adjectives *hardware* and *protocol* are used to describe the fields in the ARP packets. For example, an ARP request asks for the protocol address (an IP address in this case) corresponding to a hardware address (an Ethernet address in this case).

The *hard type* field specifies the type of hardware address. Its value is 1 for an Ethernet. *Prot type* specifies the type of protocol address being mapped. Its value is `0x0800` for IP addresses. This is purposely the same value as the type field of an Ethernet frame containing an IP datagram. (See Figure 2.1, p. 23.)

The next two 1-byte fields, *hard size* and *prot size*, specify the sizes in bytes of the hardware addresses and the protocol addresses. For an ARP request or reply for an IP address on an Ethernet they are 6 and 4, respectively.

The *op* field specifies whether the operation is an ARP request (a value of 1), ARP reply (2), RARP request (3), or RARP reply (4). (We talk about RARP in Chapter 5.) This field is required since the *frame type* field is the same for an ARP request and an ARP reply.

The next four fields that follow are the sender's hardware address (an Ethernet address in this example), the sender's protocol address (an IP address), the target hardware address, and the target protocol address. Notice there is some duplication of information: the sender's hardware address is available both in the Ethernet header and in the ARP request.

For an ARP request all the fields are filled in except the target hardware address. When a system receives an ARP request directed to it, it fills in its hardware address, swaps the two sender addresses with the two target addresses, sets the *op* field to 2, and sends the reply.

4.5 ARP Examples

In this section we'll use the `tcpdump` command to see what really happens with ARP when we execute normal TCP utilities such as Telnet. Appendix A contains additional details on the `tcpdump` program.

Normal Example

To see the operation of ARP we'll execute the `telnet` command, connecting to the discard server.

```
bsdi % arp -a                          verify ARP cache is empty
bsdi % telnet svr4 discard             connect to the discard server
Trying 140.252.13.34...
Connected to svr4.
Escape character is '^]'.
^]                                     type Control, right bracket to get Telnet client prompt
telnet> quit                           and terminate
Connection closed.
```

While this is happening we run the `tcpdump` command on another system (`sun`) with the `-e` option. This displays the hardware addresses (which in our examples are 48-bit Ethernet addresses).

```
1  0.0                  0:0:c0:6f:2d:40 ff:ff:ff:ff:ff:ff arp 60:
                        arp who-has svr4 tell bsdi
2  0.002174 (0.0022)    0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 arp 60:
                        arp reply svr4 is-at 0:0:c0:c2:9b:26
3  0.002831 (0.0007)    0:0:c0:6f:2d:40 0:0:c0:c2:9b:26 ip 60:
                        bsdi.1030 > svr4.discard: S 596459521:596459521(0)
                        win 4096 <mss 1024> [tos 0x10]
4  0.007834 (0.0050)    0:0:c0:c2:9b:26 0:0:c0:6f:2d:40 ip 60:
                        svr4.discard > bsdi.1030: S 3562228225:3562228225(0)
                        ack 596459522 win 4096 <mss 1024>
5  0.009615 (0.0018)    0:0:c0:6f:2d:40 0:0:c0:c2:9b:26 ip 60:
                        bsdi.1030 > svr4.discard: . ack 1 win 4096 [tos 0x10]
```

Figure 4.4 ARP request and ARP reply generated by TCP connection request.

Figure A.3 in Appendix A contains the raw output from `tcpdump` used for Figure 4.4. Since this is the first example of `tcpdump` output in the text, you should review that appendix to see how we've beautified the output.

We have deleted the final four lines of the `tcpdump` output that correspond to the termination of the connection (which we cover in Chapter 18), since they're not relevant to the discussion here.

In line 1 the hardware address of the source (`bsdi`) is `0:0:c0:6f:2d:40`. The destination hardware address is `ff:ff:ff:ff:ff:ff`, which is the Ethernet broadcast address. Every Ethernet interface on the cable will receive the frame and process it, as shown in Figure 4.2.

The next output field on line 1, `arp`, means the *frame type* field is `0x0806`, specifying either an ARP request or an ARP reply.

The value 60 printed after the words `arp` and `ip` on each of the five lines is the length of the Ethernet frame. Since the size of an ARP request and ARP reply is 42 bytes (28 bytes for the ARP message, 14 bytes for the Ethernet header), each frame has been padded to the Ethernet minimum: 60 bytes.

Referring to Figure 1.7, this minimum of 60 bytes starts with and includes the 14-byte Ethernet header, but does not include the 4-byte Ethernet trailer. Some books state the minimum as 64 bytes, which includes the Ethernet trailer. We purposely did not include the 14-byte Ethernet header in the minimum of 46 bytes shown in Figure 1.7, since the corresponding maximum (1500 bytes) is what's referred to as the MTU—maximum transmission unit (Figure 2.5). We use the MTU often, because it limits the size of an IP datagram, but are normally not concerned with the minimum. Most device drivers or interface cards automatically pad an Ethernet frame to the minimum size. The IP datagrams on lines 3, 4, and 5 (containing the TCP segments) are all smaller than the minimum, and have also been padded to 60 bytes.

The next field on line 1, `arp who-has`, identifies the frame as an ARP request with the IP address of `svr4` as the target IP address and the IP address of `bsdi` as the sender

IP address. `tcpdump` prints the hostnames corresponding to the IP address by default. (We'll use the `-n` option in Section 4.7 to see the actual IP addresses in an ARP request.)

From line 2 we see that while the ARP request is broadcast, the destination address of the ARP reply is `bsdi (0:0:c0:6f:2d:40)`. The ARP reply is sent directly to the requesting host; it is not broadcast.

`tcpdump` prints `arp reply` for this frame, along with the hostname and hardware address of the responder.

Line 3 is the first TCP segment requesting that a connection be established. Its destination hardware address is the destination host (`svr4`). We'll cover the details of this segment in Chapter 18.

The number printed after the line number on each line is the time (in seconds) when the packet was received by `tcpdump`. Each line other than the first also contains the time difference (in seconds) from the previous line, in parentheses. We can see in this figure that the time between sending the ARP request and receiving the ARP reply is 2.2 ms. The first TCP segment is sent 0.7 ms after this. The overhead involved in using ARP for dynamic address resolution in this example is less than 3 ms.

A final point from the `tcpdump` output is that we don't see an ARP request from `svr4` before it sends its first TCP segment (line 4). While it's possible that `svr4` already had an entry for `bsdi` in its ARP cache, normally when a system receives an ARP request, in addition to sending the ARP reply it also saves the requestor's hardware address and IP address in its own ARP cache. This is on the logical assumption that if the requestor is about to send it an IP datagram, the receiver of the datagram will probably send a reply.

ARP Request to a Nonexistent Host

What happens if the host being queried for is down or nonexistent? To see this we specify a nonexistent Internet address—the network ID and subnet ID are that of the local Ethernet, but there is no host with the specified host ID. From Figure 3.10 we see the host IDs 36 through 62 are nonexistent (the host ID of 63 is the broadcast address). We'll use the host ID 36 in this example.

```
                                        telnet to an address this time, not a hostname
bsdi % date ; telnet 140.252.13.36 ; date
Sat Jan 30 06:46:33 MST 1993
Trying 140.252.13.36...
telnet: Unable to connect to remote host: Connection timed out
Sat Jan 30 06:47:49 MST 1993            76 seconds after previous date output

bsdi % arp -a                           check the ARP cache
? (140.252.13.36) at (incomplete)
```

Figure 4.5 shows the `tcpdump` output.

```
1   0.0                  arp who-has 140.252.13.36 tell bsdi
2   5.509069 ( 5.5091)   arp who-has 140.252.13.36 tell bsdi
3  29.509745 (24.0007)   arp who-has 140.252.13.36 tell bsdi
```

Figure 4.5 ARP requests to a nonexistent host.

This time we didn't specify the `-e` option since we already know that the ARP requests are broadcast.

What's interesting here is to see the frequency of the ARP requests: 5.5 seconds after the first request, then again 24 seconds later. (We examine TCP's timeout and retransmission algorithms in more detail in Chapter 21.) The total time shown in the `tcpdump` output is 29.5 seconds. But the output from the `date` commands before and after the `telnet` command shows that the connection request from the Telnet client appears to have given up after about 75 seconds. Indeed, we'll see later that most BSD implementations set a limit of 75 seconds for a TCP connection request to complete.

In Chapter 18 when we see the sequence of TCP segments that is sent to establish the connection, we'll see that these ARP requests correspond one-to-one with the initial TCP SYN (synchronize) segment that TCP is trying to send.

Note that on the wire we never see the TCP segments. All we can see are the ARP requests. Until an ARP reply comes back, the TCP segments can't be sent, since the destination hardware address isn't known. If we ran `tcpdump` in a filtering mode, looking only for TCP data, there would have been no output at all.

ARP Cache Timeout

A timeout is normally provided for entries in the ARP cache. (In Section 4.8 we'll see that the `arp` command allows an entry to be placed into the cache by the administrator that will never time out.) Berkeley-derived implementations normally have a timeout of 20 minutes for a completed entry and 3 minutes for an incomplete entry. (We saw an incomplete entry in our previous example where we forced an ARP to a nonexistent host on the Ethernet.) These implementations normally restart the 20-minute timeout for an entry each time the entry is used.

> The Host Requirements RFC says that this timeout should occur even if the entry is in use, but most Berkeley-derived implementations do not do this—they restart the timeout each time the entry is referenced.

4.6 Proxy ARP

Proxy ARP lets a router answer ARP requests on one of its networks for a host on another of its networks. This fools the sender of the ARP request into thinking that the router is the destination host, when in fact the destination host is "on the other side" of the router. The router is acting as a proxy agent for the destination host, relaying packets to it from other hosts.

An example is the best way to describe proxy ARP. In Figure 3.10 we showed that the system `sun` was connected to two Ethernets. But we also noted that this wasn't really true, if you compare that figure with the one on the inside front cover. There is in fact a router between `sun` and the subnet 140.252.1, and this router performs proxy ARP to make it appear as though `sun` is actually on the subnet 140.252.1. Figure 4.6 shows the arrangement, with a Telebit NetBlazer, named `netb`, between the subnet and the host `sun`.

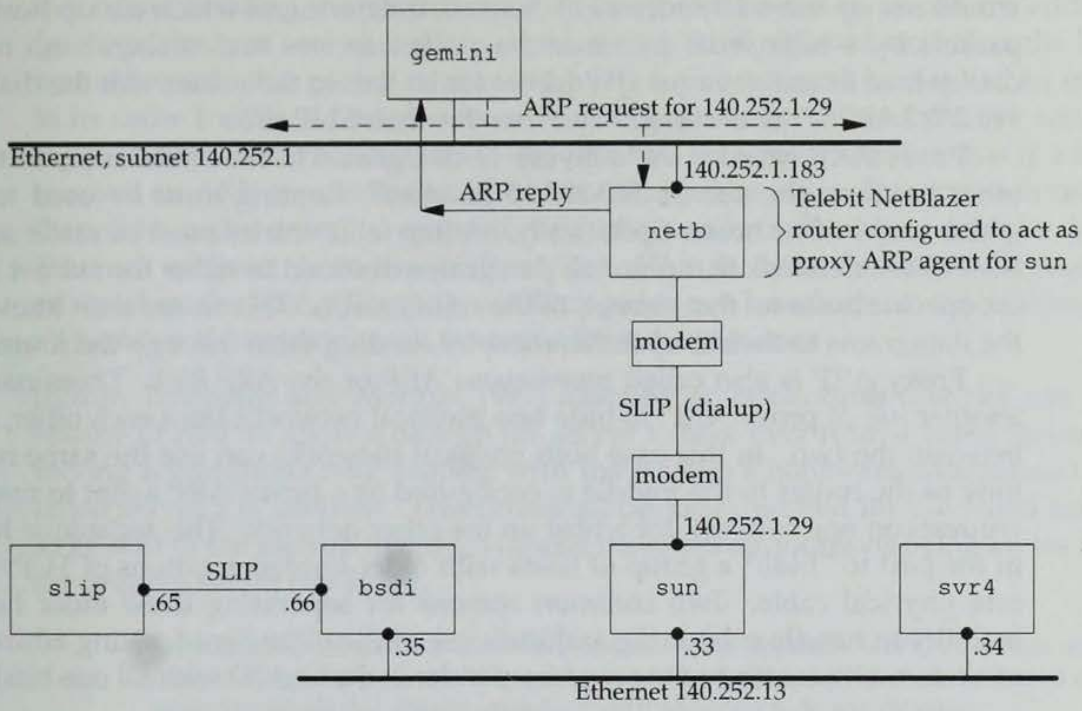


Figure 4.6 Example of proxy ARP.

When some other host on the subnet 140.252.1 (say, `gemini`) has an IP datagram to send to `sun` at address 140.252.1.29, `gemini` compares the network ID (140.252) and subnet ID (1) and since they are equal, issues an ARP request on the top Ethernet in Figure 4.6 for IP address 140.252.1.29. The router `netb` recognizes this IP address as one belonging to one of its dialup hosts, and responds with the hardware address of its Ethernet interface on the cable 140.252.1. The host `gemini` sends the IP datagram to `netb` across the Ethernet, and `netb` forwards the datagram to `sun` across the dialup SLIP link. This makes it transparent to all the hosts on the 140.252.1 subnet that host `sun` is really configured "behind" the router `netb`.

If we execute the `arp` command on the host `gemini`, after communicating with the host `sun`, we see that both IP addresses on the 140.252.1 subnet, `netb` and `sun`, map to the same hardware address. This is often a clue that proxy ARP is being used.

```
gemini % arp -a
                                    many lines for other hosts on the 140.252.1 subnet
netb (140.252.1.183) at 0:80:ad:3:6a:80
sun (140.252.1.29) at 0:80:ad:3:6a:80
```

Another detail in Figure 4.6 that we need to explain is the apparent lack of an IP address at the bottom of the router `netb` (the SLIP link). That is, why don't both ends of the dialup SLIP link have an IP address, as do both ends of the hardwired SLIP link between `bsdi` and `slip`? We noted in Section 3.8 that the destination address of the dialup SLIP link, as shown by the `ifconfig` command, was 140.252.1.183. The NetBlazer doesn't require an IP address for its end of each dialup SLIP link. (Doing so

would use up more IP addresses.) Instead, it determines which dialup host is sending it packets by which serial interface the packet arrives on, so there's no need for each dialup host to use a unique IP address for its link to the router. All the dialup hosts use 140.252.1.183 as the destination address for their SLIP link.

Proxy ARP handles the delivery of datagrams to the router `sun`, but how are the other hosts on the subnet 140.252.13 handled? Routing must be used to direct datagrams to the other hosts. Specifically, routing table entries must be made somewhere on the 140.252 network that point all datagrams destined to either the subnet 140.252.13, or the specific hosts on that subnet, to the router `netb`. This router then knows how to get the datagrams to their final destination, by sending them through the router `sun`.

Proxy ARP is also called *promiscuous ARP* or the *ARP hack*. These names are from another use of proxy ARP: to hide two physical networks from each other, with a router between the two. In this case both physical networks can use the same network ID as long as the router in the middle is configured as a proxy ARP agent to respond to ARP requests on one network for a host on the other network. This technique has been used in the past to "hide" a group of hosts with older implementations of TCP/IP on a separate physical cable. Two common reasons for separating these older hosts are their inability to handle subnetting and their use of the older broadcasting address (a host ID of all zero bits, instead of the current standard of a host ID with all one bits).

4.7 Gratuitous ARP

Another feature of ARP that we can watch is called *gratuitous ARP*. It occurs when a host sends an ARP request looking for its own IP address. This is usually done when the interface is configured at bootstrap time.

In our internet, if we bootstrap the host `bsdi` and run `tcpdump` on the host `sun`, we see the packet shown in Figure 4.7.

```
1  0.0                  0:0:c0:6f:2d:40 ff:ff:ff:ff:ff:ff arp 60:
                        arp who-has 140.252.13.35 tell 140.252.13.35
```

Figure 4.7 Example of gratuitous ARP.

(We specified the `-n` flag for `tcpdump` to print numeric dotted-decimal addresses, instead of hostnames.) In terms of the fields in the ARP request, the sender's protocol address and the target's protocol address are identical: 140.252.13.35 for host `bsdi`. Also, the source address in the Ethernet header, `0:0:c0:6f:2d:40` as shown by `tcpdump`, equals the sender's hardware address (from Figure 4.4).

Gratuitous ARP provides two features.

1. It lets a host determine if another host is already configured with the same IP address. The host `bsdi` is not expecting a reply to this request. But if a reply is received, the error message "duplicate IP address sent from Ethernet address: a:b:c:d:e:f" is logged on the console. This is a warning to the system administrator that one of the systems is misconfigured.

2. If the host sending the gratuitous ARP has just changed its hardware address (perhaps the host was shut down, the interface card replaced, and then the host was rebooted), this packet causes any other host on the cable that has an entry in its cache for the old hardware address to update its ARP cache entry accordingly. A little known fact of the ARP protocol [Plummer 1982] is that if a host receives an ARP request for an IP address that is already in the receiver's cache, then that cache entry is updated with the sender's hardware address (e.g., Ethernet address) from the ARP request. This is done for any ARP request received by the host. (Recall that ARP requests are broadcast, so this is done by all hosts on the network each time an ARP request is sent.)

   [Bhide, Elnozahy, and Morgan 1991] describe an application that can use this feature of ARP to allow a backup file server to take over from a failed server by issuing a gratuitous ARP request with the backup's hardware address and the failed server's IP address. This causes all packets destined for the failed server to be sent to the backup instead, without the client applications being aware that the original server has failed.

   Unfortunately the authors then decided against this approach, since it depends on the correct implementation of ARP on all types of clients. They obviously encountered client implementations that did not implement ARP according to its specification.

   Monitoring all the systems on the author's subnet shows that SunOS 4.1.3 and 4.4BSD both issue gratuitous ARPs when bootstrapping, but SVR4 does not.

4.8 `arp` Command

We've used this command with the `-a` flag to display all the entries in the ARP cache. Other options are provided.

The superuser can specify the `-d` option to delete an entry from the ARP cache. (This was used before running a few of the examples, to let us see the ARP exchange.)

Entries can also be added using the `-s` option. It requires a *hostname* and an Ethernet *address*: the IP address corresponding to the *hostname*, and the Ethernet *address* are added to the cache. This entry is made permanent (i.e., it won't time out from the cache) unless the keyword `temp` appears at the end of the command line.

The keyword `pub` at the end of a command line with the `-s` option causes the system to act as an ARP agent for that host. The system will answer ARP requests for the IP address corresponding to the *hostname*, replying with the specified Ethernet *address*. If the advertised *address* is the system's own, then this system is acting as a proxy ARP agent for the specified *hostname*.

4.9 Summary

ARP is a basic protocol in almost every TCP/IP implementation, but it normally does its work without the application or the system administrator being aware. The ARP cache

is fundamental to its operation, and we've used the `arp` command to examine and manipulate the cache. Each entry in the cache has a timer that is used to remove both incomplete and completed entries. The `arp` command displays and modifies entries in the ARP cache.

We followed through the normal operation of ARP along with specialized versions: proxy ARP (when a router answers ARP requests for hosts accessible on another of the router's interfaces) and gratuitous ARP (sending an ARP request for your own IP address, normally when bootstrapping).

Exercises

4.1 In the commands we typed to generate the output shown in Figure 4.4 (p. 58), what would happen if, after verifying that the local ARP cache was empty, we type the command

```
bsdi % rsh svr4 arp -a
```

to verify that the ARP cache is also empty on the destination host? (This command causes the `arp -a` command to be executed on the host `svr4`.)

4.2 Describe a test to determine if a given host handles a received gratuitous ARP request correctly.

4.3 Step 7 in Section 4.2 can take a while (milliseconds) because a packet is sent and ARP then waits for the response. How do you think ARP handles multiple datagrams that arrive from IP for the same destination address during this period?

4.4 At the end of Section 4.5 we mentioned that the Host Requirements RFC and Berkeley-derived implementations differ in their handling of the timeout of an active ARP entry. What happens if we're on a Berkeley-derived client and keep trying to contact a server host that's been taken down to replace its Ethernet board? Does this change if the server issues a gratuitous ARP when it bootstraps?

5

RARP: Reverse Address Resolution Protocol

5.1 Introduction

When a system with a local disk is bootstrapped it normally obtains its IP address from a configuration file that's read from a disk file. But a system without a disk, such as an X terminal or a diskless workstation, needs some other way to obtain its IP address.

Each system on a network has a unique hardware address, assigned by the manufacturer of the network interface. The principle of RARP is for the diskless system to read its unique hardware address from the interface card and send an RARP request (a broadcast frame on the network) asking for someone to reply with the diskless system's IP address (in an RARP reply).

While the concept is simple, the implementation is often harder than ARP for reasons described later in this chapter. The official specification of RARP is RFC 903 [Finlayson et al. 1984].

5.2 RARP Packet Format

The format of an RARP packet is almost identical to an ARP packet (Figure 4.3, p. 56). The only differences are that the *frame type* is `0x8035` for an RARP request or reply, and the *op* field has a value of 3 for an RARP request and 4 for an RARP reply.

As with ARP, the RARP request is broadcast and the RARP reply is normally unicast.

5.3 RARP Examples

In our internet we can force the host sun to bootstrap from the network, instead of its local disk. If we run an RARP server and tcpdump on the host bsdi we get the output shown in Figure 5.1. We use the -e flag to have tcpdump print the hardware addresses:

```
1  0.0                  8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                        rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42

2  0.13 (0.13)          0:0:c0:6f:2d:40 8:0:20:3:f6:42 rarp 42:
                        rarp reply 8:0:20:3:f6:42 at sun

3  0.14 (0.01)          8:0:20:3:f6:42 0:0:c0:6f:2d:40 ip 65:
                        sun.26999 > bsdi.tftp: 23 RRQ "8CFC0D21.SUN4C"
```

Figure 5.1 RARP request and reply.

The RARP request is broadcast (line 1) and the RARP reply on line 2 is unicast. The output on line 2, at sun, means the RARP reply contains the IP address for the host sun (140.252.13.33).

On line 3 we see that once sun receives its IP address, it issues a TFTP read-request (RRQ) for the file 8CFC0D21.SUN4C. (TFTP is the Trivial File Transfer Protocol. We describe it in more detail in Chapter 15.) The eight hexadecimal digits in the filename are the hex representation of the IP address 140.252.13.33 for the host sun. This is the IP address that was returned in the RARP reply. The remainder of the filename, SUN4C, indicates the type of system being bootstrapped.

tcpdump says that line 3 is an IP datagram of length 65, and not a UDP datagram (which it really is), because we are running tcpdump with the -e flag, to see the hardware-level addresses. Another point to notice in Figure 5.1 is that the length of the Ethernet frame on line 2 appears to be shorter than the minimum (which we said was 60 bytes in Section 4.5.) The reason is that we are running tcpdump on the system that is sending this Ethernet frame (bsdi). The application, rarpd, writes 42 bytes to the BSD Packet Filter device (14 bytes for the Ethernet header and 28 bytes for the RARP reply) and this is what tcpdump receives a copy of. But the Ethernet device driver pads this short frame to the minimum size for transmission (60). Had we been running tcpdump on another system, the length would have been 60.

We can see in this example that when this diskless system receives its IP address in an RARP reply, it issues a TFTP request to read a bootstrap image. At this point we won't go into additional detail about how diskless systems bootstrap themselves. (Chapter 16 describes the bootstrap sequence of a diskless X terminal using RARP, BOOTP, and TFTP.)

Figure 5.2 shows the resulting packets if there is no RARP server on the network. The destination address of each packet is the Ethernet broadcast address. The Ethernet address following who-is is the target hardware address, and the Ethernet address following tell is the sender's hardware address.

Note the frequency of the retransmissions. The first retransmission occurs after 6.55 seconds and then increases to 42.80 seconds, then goes down to 5.34 seconds, then 6.55, and then works its way back to 42.79 seconds. This continues indefinitely. If we

```
 1    0.0              8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
 2    6.55 ( 6.55)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
 3   15.52 ( 8.97)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
 4   29.32 (13.80)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
 5   52.78 (23.46)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
 6   95.58 (42.80)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
 7  100.92 ( 5.34)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
 8  107.47 ( 6.55)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
 9  116.44 ( 8.97)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
10  130.24 (13.80)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
11  153.70 (23.46)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
12  196.49 (42.79)     8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff rarp 60:
                       rarp who-is 8:0:20:3:f6:42 tell 8:0:20:3:f6:42
```

Figure 5.2 RARP requests with no RARP server on the network.

calculate the differences between each timeout interval we see a doubling effect: from 5.34 to 6.55 is 1.21 seconds, from 6.55 to 8.97 is 2.42 seconds, from 8.97 to 13.80 is 4.83 seconds, and so on. When the timeout interval reaches some limit (greater than 42.80 seconds) it's reset to 5.34 seconds.

Increasing the timeout value like this is a better approach than using the same value each time. In Figure 6.8 we'll see one wrong way to perform timeout and retransmission, and in Chapter 21 we'll see TCP's method.

5.4 RARP Server Design

While the concept of RARP is simple, the design of an RARP server is system dependent and complex. Conversely, providing an ARP server is simple, and is normally part of the TCP/IP implementation in the kernel. Since the kernel knows its IP addresses and hardware addresses, when it receives an ARP request for one of its IP addresses, it just replies with the corresponding hardware address.

RARP Servers as User Processes

The complication with an RARP server is that the server normally provides the mapping from a hardware address to an IP address for many hosts (all the diskless systems on the network). This mapping is contained in a disk file (normally `/etc/ethers` on

Unix systems). Since kernels normally don't read and parse disk files, the function of an RARP server is provided as a user process, not as part of the kernel's TCP/IP implementation.

To further complicate matters, RARP requests are transmitted as Ethernet frames with a specific Ethernet frame type field (`0x8035` from Figure 2.1.) This means an RARP server must have some way of sending and receiving Ethernet frames of this type. In Appendix A we describe how the BSD Packet Filter, Sun's Network Interface Tap, and the SVR4 Data Link Provider Interface can be used to receive these frames. Since the sending and receiving of these frames is system dependent, the implementation of an RARP server is tied to the system.

Multiple RARP Servers per Network

Another complication is that RARP requests are sent as hardware-level broadcasts, as shown in Figure 5.2. This means they are not forwarded by routers. To allow diskless systems to bootstrap even when the RARP server host is down, multiple RARP servers are normally provided on a single network (e.g., a single cable).

As the number of servers increases (to provide redundancy), the network traffic increases, since every server sends an RARP reply for every RARP request. The diskless system that sent the RARP request normally uses the first RARP reply that it receives. (We never had this problem with ARP, because only a single host sends an ARP reply.) Furthermore, there is a chance that each RARP server can try to respond at about the same time, increasing the probability of collisions on an Ethernet.

5.5 Summary

RARP is used by many diskless systems to obtain their IP address when bootstrapped. The RARP packet format is nearly identical to the ARP packet. An RARP request is broadcast, identifying the sender's hardware address, asking for anyone to respond with the sender's IP address. The reply is normally unicast.

Problems with RARP include its use of a link-layer broadcast, preventing most routers from forwarding an RARP request, and the minimal information returned: just the system's IP address. In Chapter 16 we'll see that BOOTP returns more information for the diskless system that is bootstrapping: its IP address, the name of a host to bootstrap from, and so on.

While the RARP concept is simple, the implementation of an RARP server is system dependent. Hence not all TCP/IP implementations provide an RARP server.

Exercises

5.1 Is a separate *frame type* field required for RARP? Could the same value be used for ARP and RARP `0x0806`?

5.2 With multiple RARP servers on a network, how can they prevent their responses from colliding with each on the network?

6

ICMP: Internet Control Message Protocol

6.1 Introduction

ICMP is often considered part of the IP layer. It communicates error messages and other conditions that require attention. ICMP messages are usually acted on by either the IP layer or the higher layer protocol (TCP or UDP). Some ICMP messages cause errors to be returned to user processes.

ICMP messages are transmitted within IP datagrams, as shown in Figure 6.1.

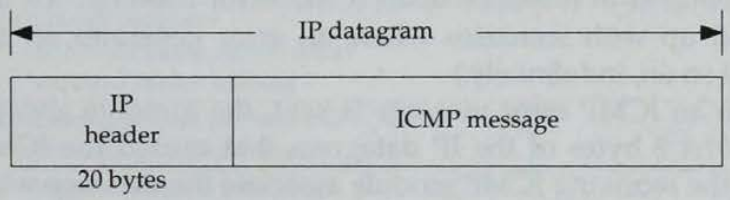


Figure 6.1 ICMP messages encapsulated within an IP datagram.

RFC 792 [Postel 1981b] contains the official specification of ICMP.

Figure 6.2 shows the format of an ICMP message. The first 4 bytes have the same format for all messages, but the remainder differs from one message to the next. We'll show the exact format of each message when we describe it.

There are 15 different values for the *type* field, which identify the particular ICMP message. Some types of ICMP messages then use different values of the *code* field to further specify the condition.

The *checksum* field covers the entire ICMP message. The algorithm used is the same as we described for the IP header checksum in Section 3.2. The ICMP checksum is required.

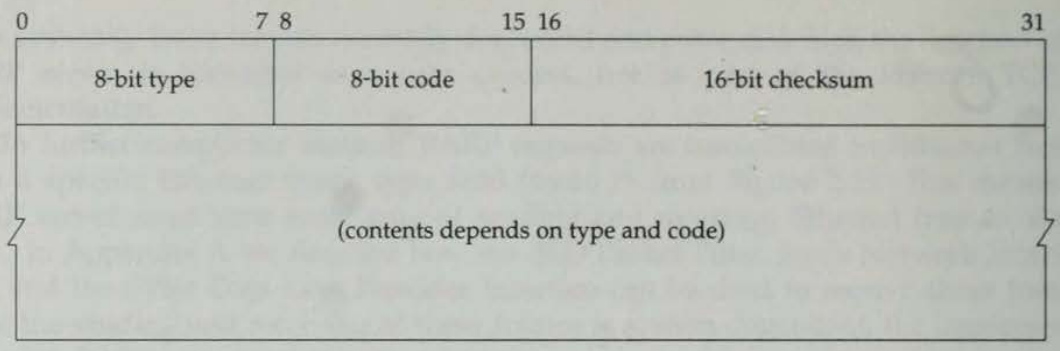


Figure 6.2 ICMP message.

In this chapter we talk about ICMP messages in general and a few in detail: address mask request and reply, timestamp request and reply, and port unreachable. We discuss the echo request and reply messages in detail with the Ping program in Chapter 7, and we discuss the ICMP messages dealing with IP routing in Chapter 9.

6.2 ICMP Message Types

Figure 6.3 lists the different ICMP message types, as determined by the *type* field and *code* field in the ICMP message.

The final two columns in this figure specify whether the ICMP message is a query message or an error message. We need to make this distinction because ICMP error messages are sometimes handled specially. For example, an ICMP error message is never generated in response to an ICMP error message. (If this were not the rule, we could end up with scenarios where an error generates an error, which generates an error, and so on, indefinitely.)

When an ICMP error message is sent, the message always contains the IP header and the first 8 bytes of the IP datagram that caused the ICMP error to be generated. This lets the receiving ICMP module associate the message with one particular protocol (TCP or UDP from the protocol field in the IP header) and one particular user process (from the TCP or UDP port numbers that are in the TCP or UDP header contained in the first 8 bytes of the IP datagram). We'll show an example of this in Section 6.5.

An ICMP error message is never generated in response to

1. An ICMP error message. (An ICMP error message may, however, be generated in response to an ICMP query message.)
2. A datagram destined to an IP broadcast address (Figure 3.9) or an IP multicast address (a class D address, Figure 1.5).
3. A datagram sent as a link-layer broadcast.
4. A fragment other than the first. (We describe fragmentation in Section 11.5.)

type	*code*	Description	Query	Error
0	0	echo reply (Ping reply, Chapter 7)	•	
3		destination unreachable:		
	0	network unreachable (Section 9.3)		•
	1	host unreachable (Section 9.3)		•
	2	protocol unreachable		•
	3	port unreachable (Section 6.5)		•
	4	fragmentation needed but don't-fragment bit set (Section 11.6)		•
	5	source route failed (Section 8.5)		•
	6	destination network unknown		•
	7	destination host unknown		•
	8	source host isolated (obsolete)		•
	9	destination network administratively prohibited		•
	10	destination host administratively prohibited		•
	11	network unreachable for TOS (Section 9.3)		•
	12	host unreachable for TOS (Section 9.3)		•
	13	communication administratively prohibited by filtering		•
	14	host precedence violation		•
	15	precedence cutoff in effect		•
4	0	source quench (elementary flow control, Section 11.11)		•
5		redirect (Section 9.5):		
	0	redirect for network		•
	1	redirect for host		•
	2	redirect for type-of-service and network		•
	3	redirect for type-of-service and host		•
8	0	echo request (Ping request, Chapter 7)	•	
9	0	router advertisement (Section 9.6)	•	
10	0	router solicitation (Section 9.6)	•	
11		time exceeded:		
	0	time-to-live equals 0 during transit (Traceroute, Chapter 8)		•
	1	time-to-live equals 0 during reassembly (Section 11.5)		•
12		parameter problem:		
	0	IP header bad (catchall error)		•
	1	required option missing		•
13	0	timestamp request (Section 6.4)	•	
14	0	timestamp reply (Section 6.4)	•	
15	0	information request (obsolete)	•	
16	0	information reply (obsolete)	•	
17	0	address mask request (Section 6.3)	•	
18	0	address mask reply (Section 6.3)	•	

Figure 6.3 ICMP message types.

5. A datagram whose source address does not define a single host. This means the source address cannot be a zero address, a loopback address, a broadcast address, or a multicast address.

These rules are meant to prevent the *broadcast storms* that have occurred in the past when ICMP errors were sent in response to broadcast packets.

6.3 ICMP Address Mask Request and Reply

The ICMP address mask request is intended for a diskless system to obtain its subnet mask (Section 3.5) at bootstrap time. The requesting system broadcasts its ICMP request. (This is similar to a diskless system using RARP to obtain its IP address at bootstrap time.) An alternative method for a diskless system to obtain its subnet mask is the BOOTP protocol, which we describe in Chapter 16. Figure 6.4 shows the format of the ICMP address mask request and reply messages.

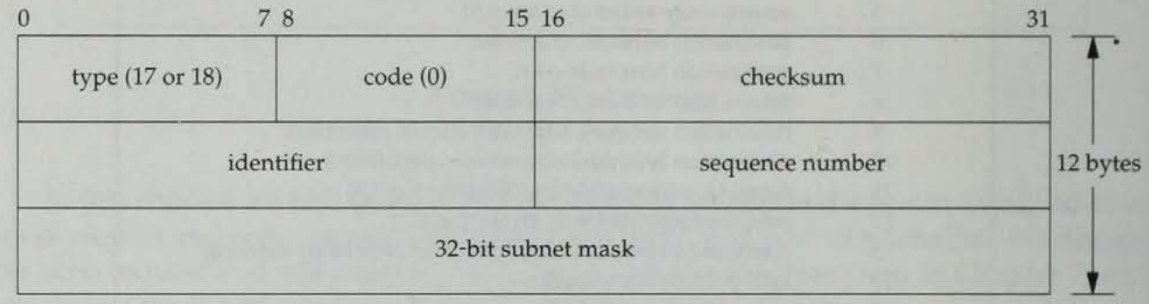


Figure 6.4 ICMP address mask request and reply messages.

The identifier and sequence number fields in the ICMP message can be set to anything the sender chooses, and these values are returned in the reply. This allows the sender to match replies with requests.

We can write a simple program (named `icmpaddrmask`) that issues an ICMP address mask request and prints all replies. Since normal usage is to send the request to the broadcast address, that's what we'll do. The destination address (140.252.13.63) is the broadcast address for the subnet 140.252.13.32 (Figure 3.12).

```
sun % icmpaddrmask 140.252.13.63
received mask = ffffffe0, from 140.252.13.33        from ourself
received mask = ffffffe0, from 140.252.13.35        from bsdi
received mask = ffff0000, from 140.252.13.34        from svr4
```

The first thing we note in this output is that the returned value from `svr4` is wrong. It appears that SVR4 is returning the general class B address mask, assuming no subnets, even though the interface on `svr4` has been configured with the correct subnet mask:

```
svr4 % ifconfig emd0
emd0: flags=23<UP,BROADCAST,NOTRAILERS>
        inet 140.252.13.34 netmask ffffffe0 broadcast 140.252.13.63
```

There is a bug in the SVR4 handling of the ICMP address mask request.

We'll watch this exchange on the host `bsdi` using `tcpdump`. The output is shown in Figure 6.5. We specify the `-e` option to see the hardware addresses.

```
1  0.0                  8:0:20:3:f6:42 ff:ff:ff:ff:ff:ff ip 60:
                        sun > 140.252.13.63: icmp: address mask request

2  0.00 (0.00)          0:0:c0:6f:2d:40 ff:ff:ff:ff:ff:ff ip 46:
                        bsdi > sun: icmp: address mask is 0xfffffe0

3  0.01 (0.01)          0:0:c0:c2:9b:26 8:0:20:3:f6:42 ip 60:
                        svr4 > sun: icmp: address mask is 0xffff0000
```

Figure 6.5 ICMP address mask request sent to broadcast address.

Note that the sending host, `sun`, receives an ICMP reply (the output line with the comment *from ourself* shown earlier), even though nothing is seen on the wire. This is a general characteristic of broadcasting: the sending host receives a copy of the broadcast packet through some internal loopback mechanism. Since by definition the term "broadcast" means *all* the hosts on the local network, it should include the sending host. (Referring to Figure 2.4 [p. 28] what is happening is that when the Ethernet driver recognizes that the destination address is the broadcast address, the packet is sent onto the network *and* a copy is made and passed to the loopback interface.)

Next, `bsdi` broadcasts the reply, while `svr4` sends the reply only to the requestor. Normally the reply should be unicast unless the source IP address of the request is 0.0.0.0, which it isn't in this example. Therefore, sending the reply to the broadcast address is a BSD/386 bug.

> The Host Requirements RFC says that a system must not send an address mask reply unless it is an authoritative agent for address masks. (To be an authoritative agent it must be specifically configured to send these replies. See Appendix E.) As we can see from this example, however, most host implementations send a reply if they get a request. Some hosts even send the wrong reply!

The final point is shown by the following example. We send an address mask request to our own IP address and to the loopback address:

```
sun % icmpaddrmask sun
received mask = ff000000, from 140.252.13.33

sun % icmpaddrmask localhost
received mask = ff000000, from 127.0.0.1
```

In both cases the returned address mask corresponds to the loopback address, the class A address 127.0.0.1. Again, referring to Figure 2.4 we see that IP datagrams sent to the host's own IP address (140.252.13.33 in this example) are actually sent to the loopback interface. The ICMP address mask reply must correspond to the subnet mask of the interface on which the request was received (since a multihomed host can have different subnet masks for each interface), and in both cases the request is received from the loopback interface.

6.4 ICMP Timestamp Request and Reply

The ICMP timestamp request allows a system to query another for the current time. The recommended value to be returned is the number of milliseconds since midnight, Coordinated Universal Time (UTC). (Older manuals refer to UTC as Greenwich Mean Time.) The nice feature of this ICMP message is that it provides millisecond resolution, whereas some other methods for obtaining the time from another host (such as the `rdate` command provided by some Unix systems) provide a resolution of seconds. The drawback is that only the time since midnight is returned—the caller must know the date from some other means.

Figure 6.6 shows the format of the ICMP timestamp request and reply messages.

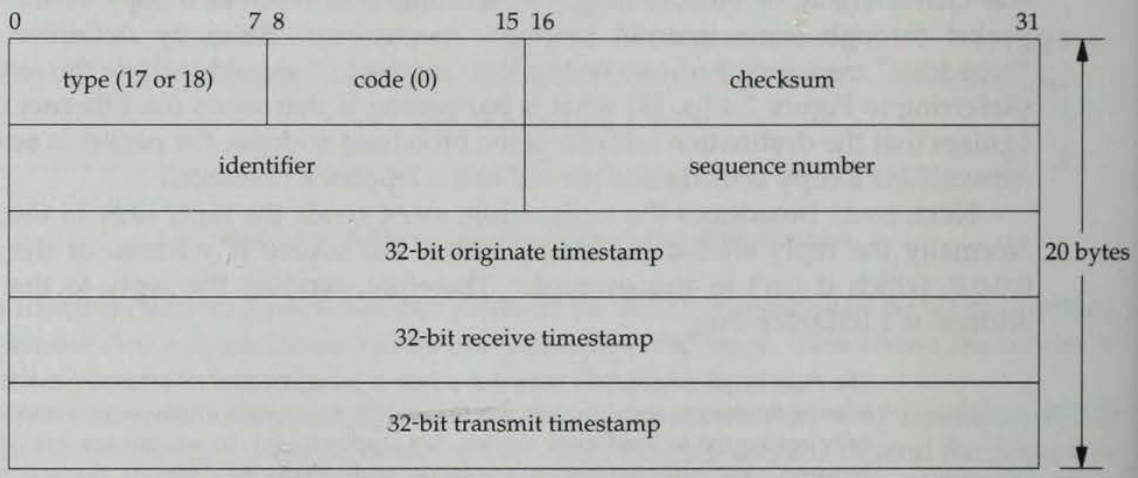


Figure 6.6 ICMP timestamp request and reply messages.

The requestor fills in the *originate* timestamp and sends the request. The replying system fills in the *receive* timestamp when it receives the request, and the *transmit* timestamp when it sends the reply. In actuality, however, most implementations set the latter two fields to the same value. (The reason for providing the three fields is to let the sender compute the time for the request to be sent, and separately compute the time for the reply to be sent.)

Examples

We can write a simple program (named `icmptime`) that sends an ICMP timestamp request to a host and prints the returned reply. We try it first on our small internet:

```
sun % icmptime bsdi
orig = 83573336, recv = 83573330, xmit = 83573330, rtt = 2 ms
difference = -6 ms

sun % icmptime bsdi
orig = 83577987, recv = 83577980, xmit = 83577980, rtt = 2 ms
difference = -7 ms
```

The program prints the three timestamps in the ICMP message: the originate (`orig`), receive (`recv`), and transmit (`xmit`) timestamps. As we can see in this and the following examples, all the hosts set the receive and transmit timestamps to the same value.

We also calculate the round-trip time (`rtt`), which is the time the reply is received minus the time the request was sent. The `difference` is the received timestamp minus the originate timestamp. Figure 6.7 shows the relationship between these values.

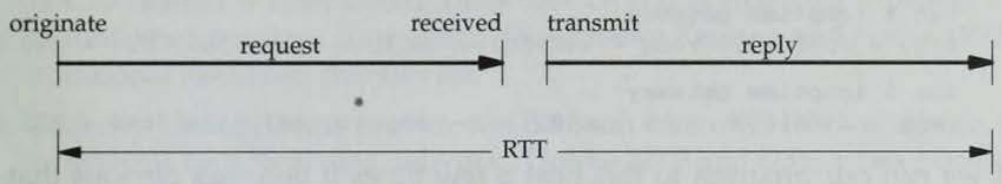


Figure 6.7 Relationship between values printed by our `icmptime` program.

If we believe the RTT and assume that one-half of the RTT is for the request, and the other half for the reply, then the sender's clock needs to be adjusted by `difference` minus one-half the RTT, to have the same time as the host being queried. In the preceding example, the clock on `bsdi` was 7 and 8 ms behind the clock on `sun`.

Since the timestamp values are the number of milliseconds past midnight, UTC, they should always be less than 86,400,000 ($24 \times 60 \times 60 \times 1000$). These examples were run just before 4:00 P.M. in a time zone that is 7 hours behind UTC, so the values being greater than 82,800,000 (2300 hours) makes sense.

If we run this program several times to the host `bsdi` we see that the final digit in the receive and transmit timestamp is always 0. This is because the software release (Version 0.9.4) only provides a 10-ms clock. (We describe this in Appendix B.)

If we run the program twice to the host `svr4` we see that the low-order three digits of the SVR4 timestamp are always 0:

```
sun % icmptime svr4
orig = 83588210, recv = 83588000, xmit = 83588000, rtt = 4 ms
difference = -210 ms

sun % icmptime svr4
orig = 83591547, recv = 83591000, xmit = 83591000, rtt = 4 ms
difference = -547 ms
```

For some reason SVR4 doesn't provide any millisecond resolution using the ICMP timestamp. This imprecision makes the calculated differences useless for subsecond adjustments.

If we try two other hosts on the 140.252.1 subnet, the results show that one clock differs from `sun`'s by 3.7 seconds, and the other by nearly 75 seconds:

```
sun % icmptime gemini
orig = 83601883, recv = 83598140, xmit = 83598140, rtt = 247 ms
difference = -3743 ms

sun % icmptime aix
orig = 83606768, recv = 83532183, xmit = 83532183, rtt = 253 ms
difference = -74585 ms
```

Another interesting example is to the router gateway (a Cisco router). It shows that when a system returns a nonstandard timestamp value (something other than milliseconds past midnight, UTC), it is supposed to turn on the high-order bit of the 32-bit timestamp. Our program detects this, and prints the receive and transmit timestamps in angle brackets (after turning off the high-order bit). Also, we can't calculate the difference between the originate and receive timestamps, since they're not the same units.

```
sun % icmptime gateway
orig = 83620811, recv = <4871036>, xmit = <4871036>, rtt = 220 ms

sun % icmptime gateway
orig = 83641007, recv = <4891232>, xmit = <4891232>, rtt = 213 ms
```

If we run our program to this host a few times it becomes obvious that the values do contain millisecond resolution and do count the number of milliseconds past some starting point, but the starting point is not midnight, UTC. (It could be a counter that's incremented every millisecond since the router was bootstrapped, for example.)

As a final example we'll compare sun's clock with a system whose clock is known to be accurate—an NTP stratum 1 server. (We say more about NTP, the Network Time Protocol, below.)

```
sun % icmptime clock.llnl.gov
orig = 83662791, recv = 83662919, xmit = 83662919, rtt = 359 ms
difference = 128 ms

sun % icmptime clock.llnl.gov
orig = 83670425, recv = 83670559, xmit = 83670559, rtt = 345 ms
difference = 134 ms
```

If we calculate the difference minus one-half the RTT, this output indicates that the clock on sun is between 38.5 and 51.5 ms fast.

Alternatives

There are other ways to obtain the time and date.

1. We described the daytime service and time service in Section 1.12. The former returns the current time and date in a human readable form, a line of ASCII characters. We can test this service using the telnet command:

```
sun % telnet bsdi daytime
Trying 140.252.13.35 ...
Connected to bsdi.
Escape character is '^]'.          first three lines output are from the Telnet client
Wed Feb  3 16:38:33 1993           here's the daytime service output
Connection closed by foreign host. this is also from the Telnet client
```

   The time server, on the other hand, returns a 32-bit binary value with the number of seconds since midnight January 1, 1900, UTC. While this provides the date, the time value is in units of a second. (The rdate command that we mentioned earlier uses the TCP time service.)

2. Serious timekeepers use the Network Time Protocol (NTP) described in RFC 1305 [Mills 1992]. This protocol uses sophisticated techniques to maintain the clocks for a group of systems on a LAN or WAN to within millisecond accuracy. Anyone interested in precise timekeeping on computers should read this RFC.
3. The Open Software Foundation's (OSF) Distributed Computing Environment (DCE) defines a Distributed Time Service (DTS) that also provides clock synchronization between computers. [Rosenberg, Kenney, and Fisher 1992] provide additional details on this service.
4. Berkeley Unix systems provide the daemon `timed`(8) to synchronize the clocks of systems on a local area network. Unlike NTP and DTS, `timed` does not work across wide area networks.

6.5 ICMP Port Unreachable Error

The last two sections looked at ICMP query messages—the address mask and timestamp queries and replies. We'll now examine an ICMP error message, the port unreachable message, a subcode of the ICMP destination unreachable message, to see the additional information returned in an ICMP error message. We'll watch this using UDP (Chapter 11).

One rule of UDP is that if it receives a UDP datagram and the destination port does not correspond to a port that some process has in use, UDP responds with an ICMP port unreachable. We can force a port unreachable using the TFTP client. (We describe TFTP in Chapter 15.)

The well-known UDP port for the TFTP server to be reading from is 69. But most TFTP client programs allow us to specify a different port using the `connect` command. We use this to specify a port of 8888:

```
bsdi % tftp
tftp> connect svr4 8888          specify the hostname and port number
tftp> get temp.foo               try to fetch a file
Transfer timed out.              about 25 seconds later
tftp> quit
```

The `connect` command saves the name of the host to contact and the port number on that host, for when we later issue the `get` command. After typing the `get` command a UDP datagram is sent to port 8888 on host `svr4`. Figure 6.8 shows the `tcpdump` output for the exchange of packets that takes place.

Before the UDP datagram can be sent to `svr4` an ARP request is sent to determine its hardware address (line 1). The ARP reply (line 2) is returned and then the UDP datagram is sent (line 3). (We have left the ARP request–reply in this `tcpdump` output to remind us that this exchange may be required before the first IP datagram is sent from one host to the other. In future output we'll delete this exchange if it's not relevant to the topic being discussed.)

```
 1   0.0                  arp who-has svr4 tell bsdi
 2   0.002050 (0.0020)    arp reply svr4 is-at 0:0:c0:c2:9b:26

 3   0.002723 (0.0007)    bsdi.2924 > svr4.8888: udp 20
 4   0.006399 (0.0037)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable

 5   5.000776 (4.9944)    bsdi.2924 > svr4.8888: udp 20
 6   5.004304 (0.0035)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable

 7  10.000887 (4.9966)    bsdi.2924 > svr4.8888: udp 20
 8  10.004416 (0.0035)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable

 9  15.001014 (4.9966)    bsdi.2924 > svr4.8888: udp 20
10  15.004574 (0.0036)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable

11  20.001177 (4.9966)    bsdi.2924 > svr4.8888: udp 20
12  20.004759 (0.0036)    svr4 > bsdi: icmp: svr4 udp port 8888 unreachable
```

Figure 6.8 ICMP port unreachable generated by TFTP request.

An ICMP port unreachable is immediately returned (line 4). But the TFTP client appears to ignore the ICMP message, sending another UDP datagram about 5 seconds later (line 5). This continues three more times before the client gives up.

Notice that the ICMP messages are exchanged between hosts, without a port number designation, while each 20-byte UDP datagram is from a specific port (2924) and to a specific port (8888).

The number 20 at the end of each UDP line is the length of the data in the UDP datagram. In this example 20 is the sum of the TFTP's 2-byte opcode, the 9-byte null terminated name `temp.foo`, and the 9-byte null terminated string `netascii`. (See Figure 15.1 for the details of the TFTP packet layout.)

If we run this same example using the `-e` option of `tcpdump` we see the exact length of each ICMP port unreachable message that's returned to the sender. This length is 70 bytes, and is allocated as shown in Figure 6.9.

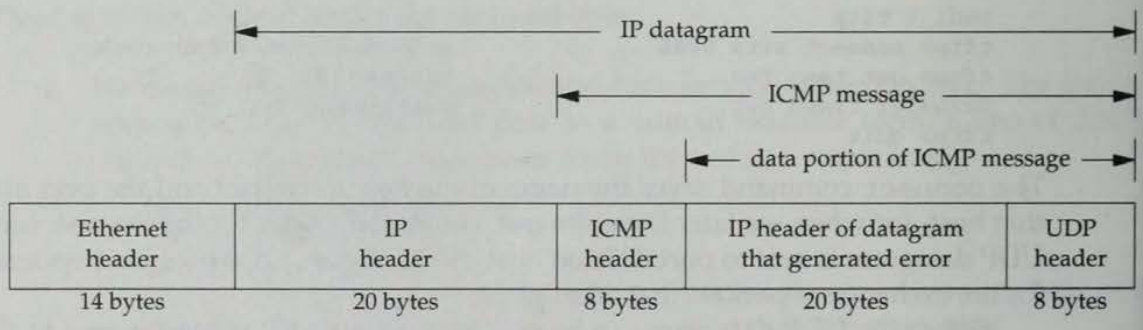


Figure 6.9 ICMP message returned for our "UDP port unreachable" example.

One rule of ICMP is that the ICMP error messages (see the final column of Figure 6.3, p. 71) must include the IP header (including any options) of the datagram that generated the error along with at least the first 8 bytes that followed this IP header. In our example, the first 8 bytes following the IP header contain the UDP header (Figure 11.2).

The important fact is that contained in the UDP header are the source and destination port numbers. It is this destination port number (8888) that caused the ICMP port unreachable to be generated. The source port number (2924) can be used by the system receiving the ICMP error to associate the error with a particular user process (the TFTP client in this example).

One reason the IP header of the datagram that caused the error is sent back is because in this IP header is the protocol field that lets ICMP know how to interpret the 8 bytes that follow (the UDP header in this example). When we look at the TCP header (Figure 17.2) we'll see that the source and destination port numbers are contained in the first 8 bytes of the TCP header.

The general format of the ICMP unreachable messages is shown in Figure 6.10.

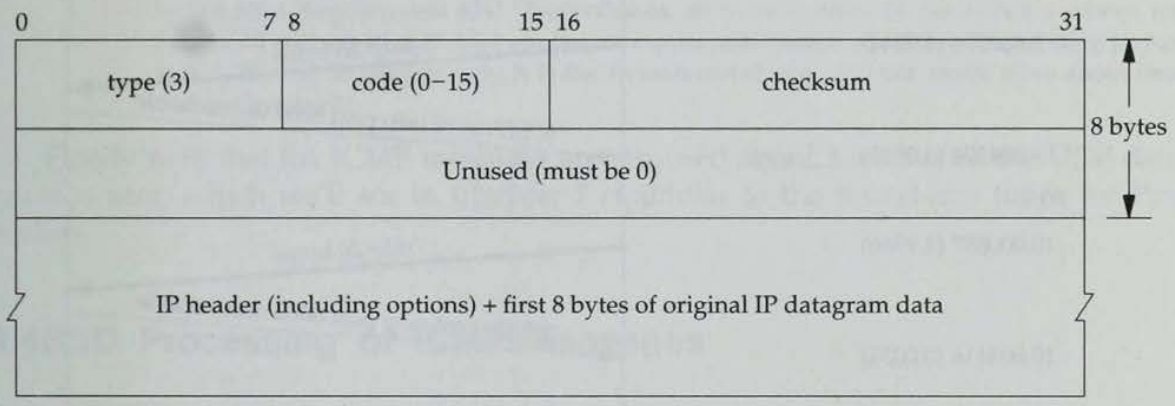


Figure 6.10 ICMP unreachable message.

In Figure 6.3 we noted that there are 16 different ICMP unreachable messages, *code*s 0 through 15. The ICMP port unreachable is *code* 3. Also, although Figure 6.10 indicates that the second 32-bit word in the ICMP message must be 0, the Path MTU Discovery mechanism (Section 2.9) allows a router to place the MTU of the outgoing interface in the low-order 16 bits of this 32-bit value, when *code* equals 4 ("fragmentation needed but the don't fragment bit is set"). We show an example of this error in Section 11.6.

> Although the rules of ICMP allow a system to return more than the first 8 bytes of the data portion of the IP datagram that caused the ICMP error, most Berkeley-derived implementations return exactly 8 bytes. The Solaris 2.2 `ip_icmp_return_data_bytes` option returns the first 64 bytes of data by default (Section E.4).

`tcpdump` Time Line

Throughout the text we'll also display the `tcpdump` output in a time line diagram as shown in Figure 6.11.

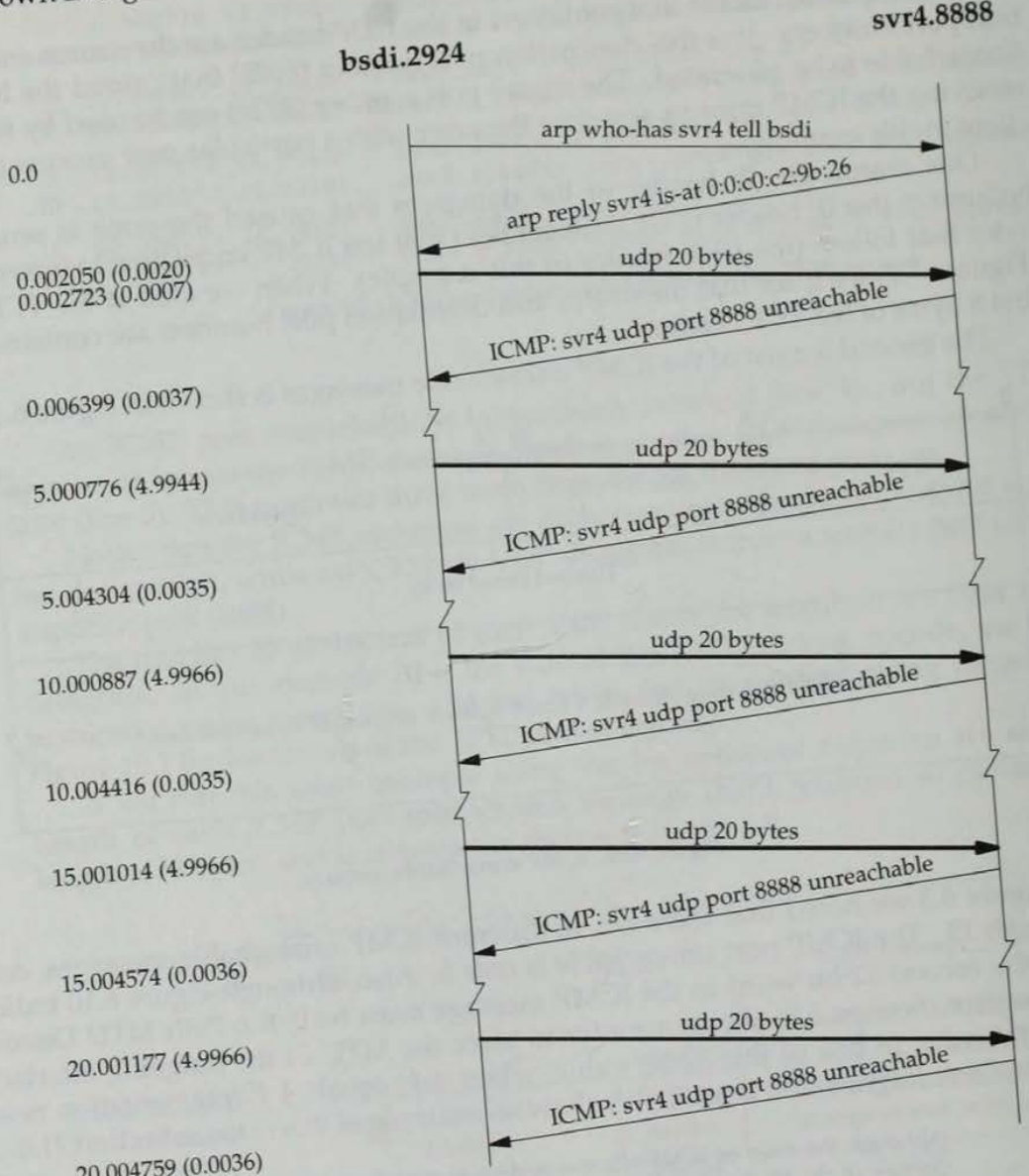


Figure 6.11 Time line of TFTP request to an invalid port.

Time increases down the page and the labels on the far left of the figure are the same time values as in our `tcpdump` output (Figure 6.8). The labels at the top are the hostnames and port numbers for each side of the time line. Be aware that the y-axis down the page is *not* exactly proportional to the time value. When there is a significant time lag, as between each 5-second retransmission in this example, we'll designate that with a squiggle on both sides of the time line. When UDP or TCP data is being transmitted, we show that packet with a thicker line.

Why does the TFTP client keep retransmitting its request when the ICMP messages are being returned? An element of network programming is occurring in which BSD systems don't notify user processes using UDP of ICMP messages that are received for that socket unless the process has issued a `connect` on that socket. The standard BSD TFTP client does not issue the `connect`, so it never receives the ICMP error notification.

Another point to notice here is the poor retransmission timeout algorithm used by this TFTP client. It just assumes that 5 seconds is adequate and retransmits every 5 seconds, for a total of 25 seconds. We'll see later that TCP has a much better algorithm.

> This old-fashioned timeout and retransmission algorithm used by the TFTP client is forbidden by the Host Requirements RFC. Nevertheless, all three systems on the author's subnet, and Solaris 2.2 still use it. AIX 3.2.2 applies an exponential backoff to its timeout, sending packets at 0, 5, 15, and 35 seconds, which is the recommended way. We talk much more about timeouts in Chapter 21.

Finally note that the ICMP messages are returned about 3.5 ms after the UDP datagram is sent, which we'll see in Chapter 7 is similar to the round-trip times for Ping replies.

6.6 4.4BSD Processing of ICMP Messages

Since ICMP covers such a wide range of conditions, from fatal errors to informational messages, each ICMP message is handled differently, even within a given implementation. Figure 6.12 is a redo of Figure 6.3, showing the handling performed by 4.4BSD for each of the possible ICMP messages.

If the final column specifies the kernel, that ICMP message is handled by the kernel. If the final column specifies "user process", then that message is passed to all user processes that have registered with the kernel to read received ICMP messages. If there are none of these user processes, the message is silently discarded. (These user processes also receive a copy of all the other ICMP messages, even those handled by the kernel, but only after the kernel has processed the message.) Some messages are completely ignored. Finally, if the final column is a string in quotes, that is the Unix error message corresponding to that condition. Some of these errors, such as TCP's handling of a source quench, we'll cover in later chapters.

type	*code*	Description	Handled by
0	0	echo reply	user process
3		destination unreachable:	
	0	network unreachable	"No route to host"
	1	host unreachable	"No route to host"
	2	protocol unreachable	"Connection refused"
	3	port unreachable	"Connection refused"
	4	fragmentation needed but DF bit set	"Message too long"
	5	source route failed	"No route to host"
	6	destination network unknown	"No route to host"
	7	destination host unknown	"No route to host"
	8	source host isolated (obsolete)	"No route to host"
	9	dest. network administratively prohibited	"No route to host"
	10	dest. host administratively prohibited	"No route to host"
	11	network unreachable for TOS	"No route to host"
	12	host unreachable for TOS	"No route to host"
	13	communication administratively prohibited	(ignored)
	14	host precedence violation	(ignored)
	15	precedence cutoff in effect	(ignored)
4	0	source quench	kernel for TCP, ignored by UDP
5		redirect:	
	0	redirect for network	kernel updates routing table
	1	redirect for host	kernel updates routing table
	2	redirect for type-of-service and network	kernel updates routing table
	3	redirect for type-of-service and host	kernel updates routing table
8	0	echo request	kernel generates reply
9	0	router advertisement	user process
10	0	router solicitation	user process
11		time exceeded:	
	0	TTL equals 0 during transit	user process
	1	TTL equals 0 during reassembly	user process
12		parameter problem:	
	0	IP header bad (catchall error)	"Protocol not available"
	1	required option missing	"Protocol not available"
13	0	timestamp request	kernel generates reply
14	0	timestamp reply	user process
15	0	information request (obsolete)	(ignored)
16	0	information reply (obsolete)	user process
17	0	address mask request	kernel generates reply
18	0	address mask reply	user process

Figure 6.12 Handling of the ICMP message types by 4.4BSD.

6.7 Summary

This chapter has been a look at the Internet Control Message Protocol, a required part of every implementation. Figure 6.3 lists all the ICMP message types, most of which we'll discuss later in the text.

We looked at the ICMP address mask request and reply and the timestamp request and reply in detail. These are typical of the request–reply messages. Both have an identifier and sequence number in the ICMP message. The sending application stores a unique value in the identifier field, to distinguish between replies for itself and replies for other processes. The sequence number field lets the client match replies with requests.

We also saw the ICMP port unreachable error, a common ICMP error. This let us examine the information returned in an ICMP error: the IP header and the next 8 bytes of the IP datagram that caused the error. This information is required by the receiver of the ICMP error, to know more about the cause of the error. Both TCP and UDP store the source and destination port numbers in the first 8 bytes of their headers for this reason.

Finally, we presented our first time line of `tcpdump` output, a presentation format we'll use in later chapters.

Exercises

6.1 At the end of Section 6.2 we listed five special conditions under which an ICMP error message is not sent. What would happen if these five conditions weren't followed and we sent a broadcast UDP datagram to an unlikely port on the local cable?

6.2 Read the Host Requirements RFC [Braden 1989a] to see if the generation of an ICMP port unreachable is a "must," "should," or "may." What section and page is this found on?

6.3 Read RFC 1349 [Almquist 1992] to see how the IP type-of-service field (Figure 3.2) should be set by ICMP.

6.4 If your system provides the `netstat` command, use it to see what types of ICMP messages are received and sent.

7

Ping Program

7.1 Introduction

The name "ping" stands for Packet InterNet Groper. The Ping program tests whether another host is reachable. The program sends an ICMP echo request message to a host, expecting an ICMP echo reply to be returned. (Figure 6.3 lists all the ICMP message types.)

Normally if you can't Ping a host, you won't be able to Telnet or FTP to that host. Conversely, if you can't Telnet to a host, Ping is often the starting point to determine what the problem is. Ping also measures the round-trip time to the host, giving us some indication of how "far away" that host is.

In this chapter we'll use Ping as a diagnostic tool and to further explore ICMP. Ping also gives us an opportunity to examine the IP record route and timestamp options. Chapter 11 of [Stevens 1990] provides the source code for the Ping program.

> Years ago we could make the unqualified statement that if we can't Ping a host, we can't Telnet or FTP to that host. With the increased awareness of security on the Internet, routers that provide access control lists, and firewall gateways, unqualified statements like this are no longer true. Reachability of a given host may depend not only on reachability at the IP layer, but also on what protocol is being used, and the port numbers involved. Ping may show a host as being unreachable, yet we might be able to Telnet to port 25 (the mail server).

7.2 Ping Program

We call the `ping` program that sends the echo requests the *client*, and the host being pinged the *server*. Most TCP/IP implementations support the Ping server directly in the kernel—the server is not a user process. (The two ICMP query services that we described in Chapter 6, the address mask and timestamp requests, are also handled directly by the kernel.)

Figure 7.1 shows the ICMP echo request and echo reply messages.

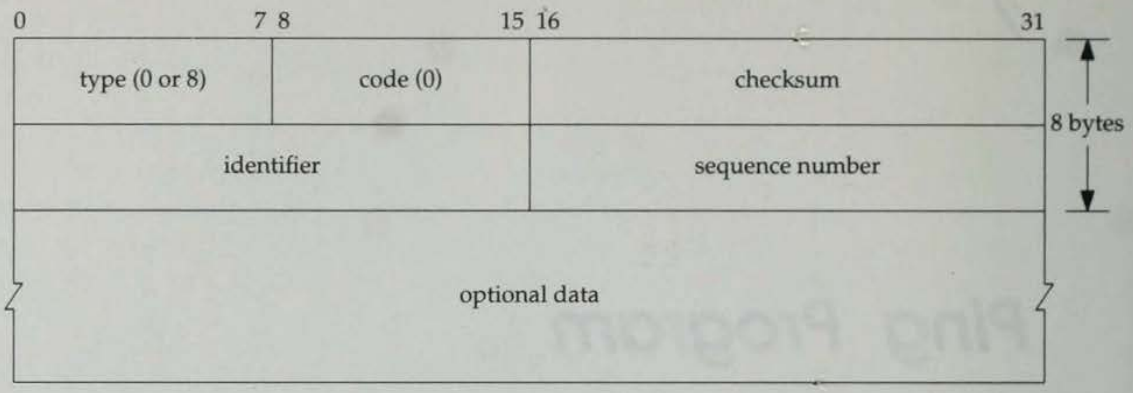


Figure 7.1 Format of ICMP message for echo request and echo reply.

As with other ICMP query messages, the server must echo the *identifier* and *sequence number* fields. Also, any optional data sent by the client must be echoed. These are presumably of interest to the client.

Unix implementations of `ping` set the *identifier* field in the ICMP message to the process ID of the sending process. This allows `ping` to identify the returned responses if there are multiple instances of `ping` running at the same time on the same host.

The *sequence number* starts at 0 and is incremented every time a new echo request is sent. `ping` prints the sequence number of each returned packet, allowing us to see if packets are missing, reordered, or duplicated. IP is a best effort datagram delivery service, so any of these three conditions can occur.

Historically the `ping` program has operated in a mode where it sends an echo request once a second, printing each echo reply that is returned. Newer implementations, however, require the `-s` option to operate this way. By default, these newer implementations send only a single echo request and output "*host* is alive" if an echo reply is received, or "no answer" if no reply is received within 20 seconds.

LAN Output

`ping` output on a LAN normally looks like the following:

```
bsdi % ping svr4
PING svr4 (140.252.13.34): 56 data bytes
64 bytes from 140.252.13.34: icmp_seq=0 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=1 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=2 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=3 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=4 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=5 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=6 ttl=255 time=0 ms
64 bytes from 140.252.13.34: icmp_seq=7 ttl=255 time=0 ms
^?                                    type interrupt key to stop
--- svr4 ping statistics ---
8 packets transmitted, 8 packets received, 0% packet loss
round-trip min/avg/max = 0/0/0 ms
```

When the ICMP echo reply is returned, the sequence number is printed, followed by the TTL, and the round-trip time is calculated. (TTL is the time-to-live field in the IP header. The current BSD `ping` program prints the received TTL each time an echo reply is received—some implementations don't do this. We examine the usage of the TTL in Chapter 8 with the `traceroute` program.)

As we can see from the output above, the echo replies were returned in the order sent (0, 1, 2, and so on).

`ping` is able to calculate the round-trip time by storing the time at which it sends the echo request in the data portion of the ICMP message. When the reply is returned it subtracts this value from the current time. Notice that on the sending system, `bsdi`, the round-trip times are all calculated as 0 ms. This is because of the low-resolution timer available to the program. The BSD/386 Version 0.9.4 system only provides a 10-ms timer. (We talk more about this in Appendix B.) We'll see later that when looking at the `tcpdump` output from this `ping` example on a system with a finer resolution clock (the Sun) the time difference between the ICMP echo request and its echo reply is just under 4 ms.

The first line of output contains the IP address of the destination host, even though we specified its name (`svr4`). This implies that the name has been converted to the IP address by a resolver. We examine resolvers and the DNS in Chapter 14. For now realize that if we type a `ping` command, and a few seconds pass before the first line of output with the IP address is printed, this is the time required for the DNS to determine the IP address corresponding to the hostname.

Figure 7.2 shows the `tcpdump` output for this example.

```
 1  0.0                  bsdi > svr4: icmp: echo request
 2  0.003733 (0.0037)    svr4 > bsdi: icmp: echo reply

 3  0.998045 (0.9943)    bsdi > svr4: icmp: echo request
 4  1.001747 (0.0037)    svr4 > bsdi: icmp: echo reply

 5  1.997818 (0.9961)    bsdi > svr4: icmp: echo request
 6  2.001542 (0.0037)    svr4 > bsdi: icmp: echo reply

 7  2.997610 (0.9961)    bsdi > svr4: icmp: echo request
 8  3.001311 (0.0037)    svr4 > bsdi: icmp: echo reply

 9  3.997390 (0.9961)    bsdi > svr4: icmp: echo request
10  4.001115 (0.0037)    svr4 > bsdi: icmp: echo reply

11  4.997201 (0.9961)    bsdi > svr4: icmp: echo request
12  5.000904 (0.0037)    svr4 > bsdi: icmp: echo reply

13  5.996977 (0.9961)    bsdi > svr4: icmp: echo request
14  6.000708 (0.0037)    svr4 > bsdi: icmp: echo reply

15  6.996764 (0.9961)    bsdi > svr4: icmp: echo request
16  7.000479 (0.0037)    svr4 > bsdi: icmp: echo reply
```

Figure 7.2 `ping` output across a LAN.

The time between sending the echo request and receiving the echo reply is always 3.7 ms. We can also see that echo requests are sent approximately 1 second apart.

Often the first round-trip time is larger than the rest. This occurs if the destination's hardware address isn't in the ARP cache of the sender. As we saw in Chapter 4, sending an ARP request and getting the ARP reply can take a few milliseconds before the first echo request can be sent. The following example shows this:

```
sun % arp -a                                    make sure ARP cache is empty
sun % ping svr4
PING svr4: 56 data bytes
64 bytes from svr4 (140.252.13.34): icmp_seq=0. time=7. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=1. time=4. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=2. time=4. ms
64 bytes from svr4 (140.252.13.34): icmp_seq=3. time=4. ms
^?                                              type interrupt key to stop
----svr4 PING Statistics----
4 packets transmitted, 4 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 4/4/7
```

The additional 3 ms in the first RTT is probably for the ARP request and reply.

This example was run on the host sun, which provides a timer with microsecond resolution, but the ping program prints the round-trip times with only millisecond resolution. The earlier example, run under BSD/386 Version 0.9.4, printed the round-trip times as 0 ms, since the available timer provided only 10-ms accuracy. The following output is from BSD/386 Version 1.0, which provides a timer with microsecond resolution and a version of ping that prints the higher resolution.

```
bsdi % ping svr4
PING svr4 (140.252.13.34): 56 data bytes
64 bytes from 140.252.13.34: icmp_seq=0 ttl=255 time=9.304 ms
64 bytes from 140.252.13.34: icmp_seq=1 ttl=255 time=6.089 ms
64 bytes from 140.252.13.34: icmp_seq=2 ttl=255 time=6.079 ms
64 bytes from 140.252.13.34: icmp_seq=3 ttl=255 time=6.096 ms
^?                                              type interrupt key to stop
--- svr4 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 6.079/6.880/9.304 ms
```

WAN Output

On a wide area network the results can be quite different. The following example was captured on a weekday afternoon, a time when the Internet is normally busy:

```
gemini % ping vangogh.cs.berkeley.edu
PING vangogh.cs.berkeley.edu: 56 data bytes
64 bytes from (128.32.130.2): icmp_seq=0. time=660. ms
64 bytes from (128.32.130.2): icmp_seq=5. time=1780. ms
64 bytes from (128.32.130.2): icmp_seq=7. time=380. ms
64 bytes from (128.32.130.2): icmp_seq=8. time=420. ms
64 bytes from (128.32.130.2): icmp_seq=9. time=390. ms
64 bytes from (128.32.130.2): icmp_seq=14. time=110. ms
64 bytes from (128.32.130.2): icmp_seq=15. time=170. ms
64 bytes from (128.32.130.2): icmp_seq=16. time=100. ms
^?                                              type interrupt key to stop
```

```
----vangogh.CS.Berkeley.EDU PING Statistics----
17 packets transmitted, 8 packets received, 52% packet loss
round-trip (ms)  min/avg/max = 100/501/1780
```

Either the echo requests or the echo replies for sequence numbers 1, 2, 3, 4, 6, 10, 11, 12, and 13 were lost somewhere. Note also the large variance in the round-trip times. (This high packet loss rate of 52% is an anomaly. This is not normal for the Internet, even on a weekday afternoon.)

It is also possible across WANs to see packets duplicated (the same sequence number printed two or more times), and to see packets reordered (sequence number $N+1$ printed before sequence number N).

Hardwired SLIP Links

Let's look at the round-trip times encountered over SLIP links, since they often run at slow asynchronous speeds, such as 9600 bits/sec or less. Recall our serial line throughput calculations in Section 2.10. For this example we'll set the speed of the hardwired SLIP link between hosts `bsdi` and `slip` to 1200 bits/sec.

We can estimate the round-trip time as follows. First, notice from the example Ping output shown earlier that by default it sends 56 bytes of data in the ICMP message. With a 20-byte IP header and an 8-byte ICMP header this gives a total IP datagram size of 84 bytes. (We can verify this by running `tcpdump -e` and seeing the Ethernet frame sizes.) Also, from Section 2.4 we know that at least two additional bytes are added: the END byte at the beginning and end of the datagram. It's also possible for additional bytes to be added by the SLIP framing, but that depends on the value of each byte in the datagram. At 1200 bits/sec with 8 bits per byte, 1 start bit, and 1 stop bit, the rate is 120 bytes per second, or 8.33 ms per byte. Our estimate is then $(86 \times 8.33 \times 2)$, or 1433 ms. (The multiplier of 2 is because we are calculating the round-trip time.)

The following output verifies our calculation:

```
svr4 % ping -s slip
PING slip: 56 data bytes
64 bytes from slip (192.42.62.1): icmp_seq=0. time=1480. ms
64 bytes from slip (192.42.62.1): icmp_seq=1. time=1480. ms
64 bytes from slip (192.42.62.1): icmp_seq=2. time=1480. ms
64 bytes from slip (192.42.62.1): icmp_seq=3. time=1480. ms
^?
----slip PING Statistics----
5 packets transmitted, 4 packets received, 20% packet loss
round-trip (ms)  min/avg/max = 1480/1480/1480
```

(The `-s` option is required for SVR4 to send one request every second.) The round-trip time is almost 1.5 seconds but the program is still sending out each ICMP echo request at 1-second intervals. This means there are two outstanding echo requests (sent at time 0 and time 1) before the first reply comes back (at time 1.480). That's also why the summary line says one packet has been lost. It really hasn't been lost, it's probably still on its way back.

We'll return to this slow SLIP link in Chapter 8 when we examine the `traceroute` program.

Dialup SLIP Links

The conditions change with a dialup SLIP link since we now have modems on each end of the link. The modems being used between the systems `sun` and `netb` provide what is called V.32 modulation (9600 bits/sec), V.42 error control (also called LAP-M), and V.42bis data compression. This means that our simple calculations, which were fairly accurate for a hardwired link where we knew all the parameters, become less accurate.

Numerous factors are at work. The modems introduce some latency. The size of the packets may decrease with the data compression, but the size may then increase to a multiple of the packet size used by the error control protocol. Also the receiving modem can't release received data bytes until the cyclic redundancy character (the checksum) has been verified. Finally, we're dealing with a computer's asynchronous serial interface on each end, and many operating systems read these interfaces only at fixed intervals, or when a certain number of characters have been received.

As an example, we ping the host `gemini` from the host `sun`:

```
sun % ping gemini
PING gemini: 56 data bytes
64 bytes from gemini (140.252.1.11): icmp_seq=0. time=373. ms
64 bytes from gemini (140.252.1.11): icmp_seq=1. time=360. ms
64 bytes from gemini (140.252.1.11): icmp_seq=2. time=340. ms
64 bytes from gemini (140.252.1.11): icmp_seq=3. time=320. ms
64 bytes from gemini (140.252.1.11): icmp_seq=4. time=330. ms
64 bytes from gemini (140.252.1.11): icmp_seq=5. time=310. ms
64 bytes from gemini (140.252.1.11): icmp_seq=6. time=290. ms
64 bytes from gemini (140.252.1.11): icmp_seq=7. time=300. ms
64 bytes from gemini (140.252.1.11): icmp_seq=8. time=280. ms
64 bytes from gemini (140.252.1.11): icmp_seq=9. time=290. ms
64 bytes from gemini (140.252.1.11): icmp_seq=10. time=300. ms
64 bytes from gemini (140.252.1.11): icmp_seq=11. time=280. ms
----gemini PING Statistics----
12 packets transmitted, 12 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 280/314/373
```

Note that the first RTT is not a multiple of 10 ms, but every other line is. If we run this numerous times, we see this property every time. (This is not caused by the resolution of the clock on the host `sun`, because we know that its clock provides millisecond resolution from the tests we run in Appendix B.)

Also note that the first RTT is larger than the next, and they keep decreasing, and then they range between 280 and 300 ms. If we let it run for a minute or two, the RTTs stay in this range, never going below 260 ms. If we calculate the expected RTT at 9600 bits/sec (Exercise 7.2) we get 180 ms, so our observed values are about 1.5 times the expected value.

If we run `ping` for 60 seconds and look at the average RTT it calculates, we find that with V.42 and V.42bis our average is 277 ms. (This is better than the average printed for our preceding example, because we ran it longer to amortize the longer RTTs at the beginning.) If we turn off just the V.42bis data compression our average is 330 ms. If we turn off the V.42 error control (which also turns off the V.42bis data compression) our average is 300 ms. These modem parameters do affect the RTTs, and using the error control and data compression appears to be the best.

7.3 IP Record Route Option

The `ping` program gives us an opportunity to look at the IP record route (RR) option. Most versions of `ping` provide the `-R` option that enables the record route feature. It causes `ping` to set the IP RR option in the outgoing IP datagram (which contains the ICMP echo request message). This causes every router that handles the datagram to add its IP address to a list in the options field. When the datagram reaches the final destination, the list of IP addresses should be copied into the outgoing ICMP echo reply, and all the routers on the return path also add their IP addresses to the list. When `ping` receives the echo reply it prints the list of IP addresses.

As simple as this sounds, there are pitfalls. Generation of the RR option by the source host, processing of the RR option by the intermediate routers, and reflection of the incoming RR list in an ICMP echo request into the outgoing ICMP echo reply are all optional features. Fortunately, most systems today do support these optional features, but some systems don't reflect the IP list.

The biggest problem, however, is the limited room in the IP header for the list of IP addresses. We saw in Figure 3.1 (p. 34) that the *header length* in the IP header is a 4-bit field, limiting the entire IP header to 15 32-bit words (60 bytes). Since the fixed size of the IP header is 20 bytes, and the RR option uses 3 bytes for overhead (which we describe below), this leaves 37 bytes (60 – 20 – 3) for the list, allowing up to nine IP addresses. In the early days of the ARPANET, nine IP addresses seemed like a lot, but since this is a round-trip list (in the case of the `-R` option for `ping`), it's of limited use today. (In Chapter 8 we'll look at the Traceroute tool for determining the route followed by a datagram.) Despite these shortcomings, the record route option works and provides an opportunity to look in detail at the handling of IP options.

Figure 7.3 shows the general format of the RR option in the IP datagram.

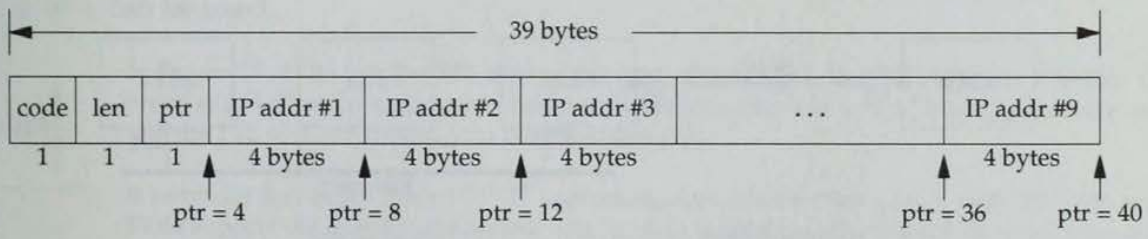


Figure 7.3 General format of record route option in IP header.

Code is a 1-byte field specifying the type of IP option. For the RR option its value is 7. *Len* is the total number of bytes of the RR option, which in this case is 39. (Although it's possible to specify an RR option with less than the maximum size, `ping` always provides a 39-byte option field, to record up to nine IP addresses. Given the limited room in the IP header for options, it doesn't make sense to specify a size less than the maximum.)

Ptr is called the pointer field. It is a 1-based index into the 39-byte option of where to store the next IP address. Its minimum value is 4, which is the pointer to the first IP address. As each IP address is recorded into the list, the value of *ptr* becomes 8, 12, 16, up to 36. After the ninth address is recorded *ptr* becomes 40, indicating the list is full.

When a router (which by definition is multihomed) records its IP address in the list, which IP address is recorded? It could be the address of the incoming interface or the outgoing interface. RFC 791 [Postel 1981a] specifies that the router records the outgoing IP address. We'll see that when the originating host (the host running `ping`) receives the ICMP echo reply with the RR option enabled, it also records its incoming IP address in the list.

Normal Example

Let's run an example of the RR option with the `ping` program. We'll run `ping` on the host `svr4` to the host `slip`. One intermediate router (`bsdi`) will handle the datagram. The following output is from `svr4`:

```
svr4 % ping -R slip
PING slip (140.252.13.65): 56 data bytes
64 bytes from 140.252.13.65: icmp_seq=0 ttl=254 time=280 ms
RR:     bsdi (140.252.13.66)
        slip (140.252.13.65)
        bsdi (140.252.13.35)
        svr4 (140.252.13.34)
64 bytes from 140.252.13.65: icmp_seq=1 ttl=254 time=280 ms (same route)
64 bytes from 140.252.13.65: icmp_seq=2 ttl=254 time=270 ms (same route)
^?
--- slip ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 270/276/280 ms
```

Figure 7.4 shows the four hops that the packets take (two in each direction), and which hop adds which IP address to the RR list.

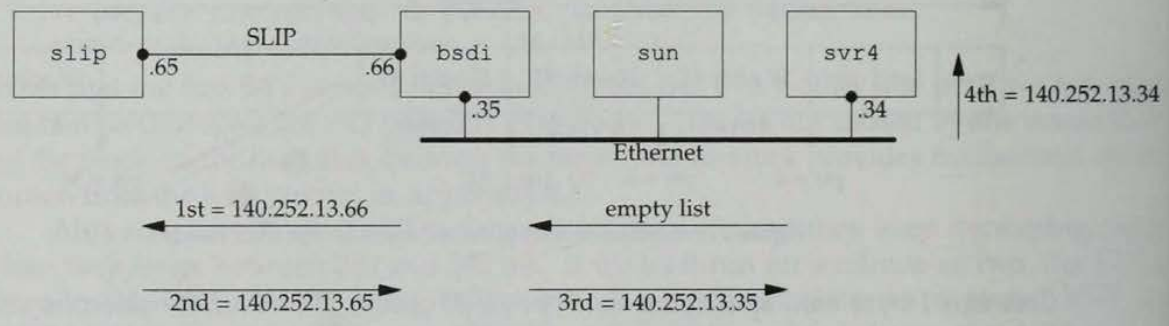


Figure 7.4 `ping` with record route option.

The router `bsdi` adds a different IP address to the list in each direction. It always adds the IP address of the outgoing interface. We can also see that when the ICMP echo reply reaches the originating system (`svr4`) it adds the IP address of the incoming interface to the list.

We can also watch this exchange of packets from the host sun, running tcpdump with its -v option (to see the IP options). Figure 7.5 shows the output.

```
1  0.0                    svr4 > slip: icmp: echo request (ttl 32, id 35835,
                          optlen=40 RR{39}= RR{#0.0.0.0/0.0.0.0/0.0.0.0/
                          0.0.0.0/ 0.0.0.0/0.0.0.0/0.0.0.0/0.0.0.0/0.0.0.0} EOL)

2  0.267746 (0.2677)      slip > svr4: icmp: echo reply (ttl 254, id 1976,
                          optlen=40 RR{39}= RR{140.252.13.66/140.252.13.65/
                          140.252.13.35/#0.0.0.0/0.0.0.0/0.0.0.0/0.0.0.0/
                          0.0.0.0/0.0.0.0} EOL)
```

Figure 7.5 tcpdump output of record route option.

The output optlen=40 indicates there are 40 bytes of option space in the IP header. (Recall that the length of the IP header must be a multiple of 4 bytes.) RR{39} means the record route option is present, and its length field is 39. The list of nine IP addresses is then shown, with a pound sign (#) indicating which IP address is pointed to by the *ptr* field in the RR option header. Since we are watching these packets on the host sun (see Figure 7.4) we only see the ICMP echo request with the empty list, and the ICMP echo reply with three addresses in the list. We have deleted the remaining lines in the tcpdump output, since they are nearly identical to what we show in Figure 7.5.

The notation EOL at the end of the record route information indicates the IP option "end of list" value appeared. The EOL option has a value of 0. What's happening is that 39 bytes of RR data are in the 40 bytes of option space in the IP header. Since the option space is set to 0 before the datagram is sent, this final byte of 0 that follows the 39 bytes of RR data is interpreted as an EOL. That is what we want to have happen. If there are multiple options in the option field of the IP header, and pad bytes are needed before the next option starts, the other special character NOP ("no operation"), with a value of 1, can be used.

> In Figure 7.5, SVR4 sets the TTL field of the echo request to 32, and BSD/386 sets it to 255. (It prints as 254 since the router bsdi has already decremented it by one.) Newer systems are setting the TTL of ICMP messages to the maximum (255).
>
> It turns out that of the three TCP/IP implementations used by the author, both BSD/386 and SVR4 support the record route option. That is, they correctly update the RR list when forwarding a datagram, and they correctly reflect the RR list from an incoming ICMP echo request to the outgoing ICMP echo reply. SunOS 4.1.3, however, updates the RR list when forwarding a datagram, but does not reflect the RR list. Solaris 2.x corrects this problem.

Abnormal Output

The following example was seen by the author and provides a starting point for our description of the ICMP redirect message in Chapter 9. We ping the host aix on the 140.252.1 subnet (accessible through the dialup SLIP connection on the host sun) with the record route option. We get the following output, when run on the host slip:

```
slip % ping -R aix
PING aix (140.252.1.92): 56 data bytes
64 bytes from 140.252.1.92: icmp_seq=0 ttl=251 time=650 ms
RR:     bsdi (140.252.13.35)
        sun (140.252.1.29)
        netb (140.252.1.183)
        aix (140.252.1.92)
        gateway (140.252.1.4)          why is this router used?
        netb (140.252.1.183)
        sun (140.252.13.33)
        bsdi (140.252.13.66)
        slip (140.252.13.65)
64 bytes from aix: icmp_seq=1 ttl=251 time=610 ms (same route)
64 bytes from aix: icmp_seq=2 ttl=251 time=600 ms (same route)
^?
--- aix ping statistics ---
4 packets transmitted, 3 packets received, 25% packet loss
round-trip min/avg/max = 600/620/650 ms
```

We could have run this example from the host `bsdi`. We chose to run it from `slip` to see all nine IP addresses in the RR list used.

The puzzle in this output is why the outgoing datagram (the ICMP echo request) went directly from `netb` to `aix`, but the return (the ICMP echo reply) went from `aix`, through the router `gateway`, before going to `netb`. What we're seeing here is a feature of IP routing that we describe below. Figure 7.6 shows the path of the datagrams.

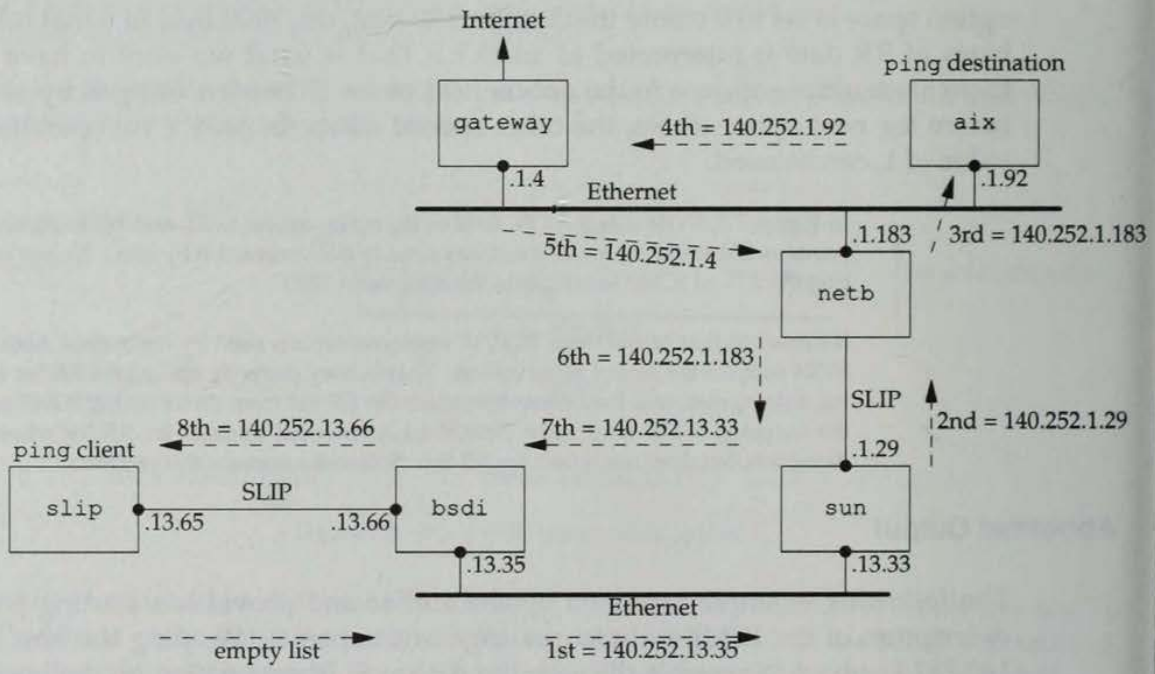


Figure 7.6 `ping` with record route, showing IP routing feature.

The problem is that `aix` does not know to send IP datagrams destined for the subnet 140.252.13 to `netb`. Instead, `aix` has a default entry in its routing table that tells it to

send all datagrams to the router `gateway` if it doesn't have a particular route for the destination. The router `gateway` has more routing knowledge than any of the hosts on the 140.252.1 subnet. (There are more than 150 hosts on this Ethernet and instead of running a routing daemon on every one, each has a "default" entry that points to the router `gateway`.)

An unanswered question here is why doesn't `gateway` send an ICMP redirect (Section 9.5) to `aix` to update its routing table? For some reason (perhaps that the datagram generating the redirect is an ICMP echo request message) the redirect is not generated. But if we use Telnet and connect to the daytime server on `aix`, the ICMP redirect is generated, and the routing table on `aix` is updated. If we then execute `ping` with the record route option enabled, the route shows that the datagrams go from `netb` to `aix` and back to `netb`, without the extra hop to the router `gateway`. We'll look at these ICMP redirects in more detail in Section 9.5.

7.4 IP Timestamp Option

The IP timestamp option is similar to the record route option. Figure 7.7 shows the format of the IP timestamp option (compare with Figure 7.3).

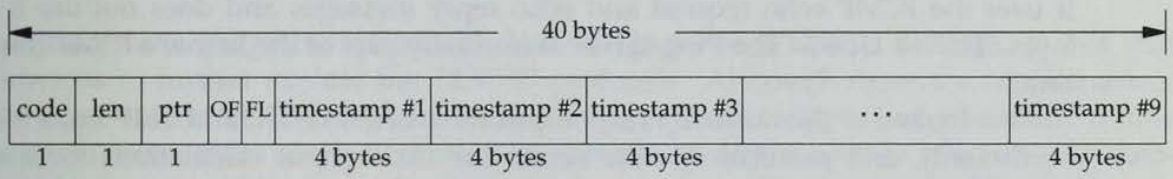


Figure 7.7 General format of timestamp option in IP header.

The *code* field is `0x44` for the timestamp option. The two fields *len* and *ptr* are the same as for the record route option: the total length of the option (normally 36 or 40) and a pointer to the next available entry (5, 9, 13, etc.).

The next two fields are 4-bit values: *OF* is the *overflow* field and *FL* is a *flags* field. The operation of the timestamp option is driven by the *flags* field, as shown in Figure 7.8.

flags	Description
0	Record only timestamps. This is what we show in Figure 7.7.
1	Each router records its IP address and its timestamp. There is room for only four of these pairs in the options list.
3	The sender initializes the options list with up to four pairs of IP addresses and a 0 timestamp. A router records its timestamp only if the next IP address in the list matches the router's.

Figure 7.8 Meaning of the *flags* value for timestamp option.

If a router can't add a timestamp because there's no room left, it just increments the *overflow* field.

The preferred value for the timestamps is the number of milliseconds past midnight, UTC, similar to the ICMP timestamp request and reply (Section 6.4). If this format is not available to a router, it can insert whatever time representation that is uses, but must then turn on the high-order bit of the timestamp to indicate the nonstandard value.

Given the limitations that we encountered with the record route option, things get worse with the timestamp option. If we record both IP addresses and timestamps (a *flags* of 1), we can store only four of these pairs. Recording only timestamps is next to useless because we have no indication regarding which timestamp corresponds to which router (unless we have a fixed topology that never changes). A *flags* of 3 is better, as we can then select which routers insert their timestamp. A more fundamental problem is that you probably have no control over how accurate the timestamp is at any given router. This makes it fruitless to try to measure hop times between routers using this IP option. We'll see that the traceroute program (Chapter 8) provides a better way of measuring hop times between routers.

7.5 Summary

The `ping` program is the basic connectivity test between two systems running TCP/IP. It uses the ICMP echo request and echo reply messages and does not use a transport layer (TCP or UDP). The Ping server is normally part of the kernel's ICMP implementation.

We looked at the normal `ping` output for a LAN, WAN, and SLIP links (dialup and hardwired), and performed some serial line throughput calculations for a dedicated SLIP link. `ping` also let us examine and use the IP record route option. We used this IP option to see how default routes are often used, and will return to this topic in Chapter 9. We also looked at the IP timestamp option, but it is of limited practical use.

Exercises

7.1 Draw a time line for the `ping` output for the SLIP link in Section 7.2.

7.2 Calculate the RTT if the SLIP link between `bsdi` and `slip` is set to 9600 bits/sec. Assume the default of 56 bytes of data.

7.3 The current BSD `ping` program allows us to specify a pattern for the data portion of the ICMP message. (The first 8 bytes of the data portion are not filled with the pattern, since the time at which the packet is sent is stored here.) If we specify a pattern of `0xc0`, recalculate the answer to the previous exercise. (*Hint*: Reread Section 2.4.)

7.4 Does the use of compressed SLIP (CSLIP, Section 2.5) affect the `ping` times that we observed in Section 7.2?

7.5 Examine Figure 2.4 (p. 28). Do you expect any difference between a ping of the loopback address, versus a ping of the host's Ethernet address?

8

Traceroute Program

8.1 Introduction

The Traceroute program, written by Van Jacobson, is a handy debugging tool that allows us to further explore the TCP/IP protocols. Although there are no guarantees that two consecutive IP datagrams from the same source to the same destination follow the same route, most of the time they do. Traceroute lets us see the route that IP datagrams follow from one host to another. Traceroute also lets us use the IP source route option.

> The manual page states: "Implemented by Van Jacobson from a suggestion by Steve Deering. Debugged by a cast of thousands with particularly cogent suggestions or fixes from C. Philip Wood, Tim Seaver, and Ken Adelman."

8.2 Traceroute Program Operation

In Section 7.3 we described the IP record route option (RR). Why wasn't this used instead of developing a new application? There are three reasons. First, historically not all routers have supported the record route option, making it unusable on certain paths. (Traceroute doesn't require any special or optional features at any intermediate routers.)

Second, record route is normally a one-way option. The sender enables the option and the receiver has to fetch all the values from the received IP header and somehow return them to the sender. In Section 7.3 we saw that most implementations of the Ping server (the ICMP echo reply function within the kernel) reflect an incoming RR list, but this doubles the number of IP addresses recorded (the outgoing path and the return path), which runs into the limit described in the next paragraph. (Traceroute requires only a working UDP module at the destination—no special server application is required.)

The third and major reason is that the room allocated for options in the IP header isn't large enough today to handle most routes. There is room for only nine IP addresses in the IP header options field. In the old days of the ARPANET this was adequate, but it is far too small nowadays.

Traceroute uses ICMP and the TTL field in the IP header. The TTL field (time-to-live) is an 8-bit field that the sender initializes to some value. The recommended initial value is specified in the Assigned Numbers RFC and is currently 64. Older systems would often initialize it to 15 or 32. We saw in some of the Ping examples in Chapter 7 that ICMP echo replies are often sent with the TTL set to its maximum value of 255.

Each router that handles the datagram is required to decrement the TTL by either one or the number of seconds that the router holds onto the datagram. Since most routers hold a datagram for less than a second, the TTL field has effectively become a hop counter, decremented by one by each router.

> RFC 1009 [Braden and Postel 1987] required a router that held a datagram for more than 1 second to decrement the TTL by the number of seconds. Few routers implemented this requirement. The new Router Requirements RFC [Almquist 1993] makes this optional, allowing a router to treat the TTL as just a hop count.

The purpose of the TTL field is to prevent datagrams from ending up in infinite loops, which can occur during routing transients. For example, when a router crashes or when the connection between two routers is lost, it can take the routing protocols some time (from seconds to a few minutes) to detect the lost route and work around it. During this time period it is possible for the datagram to end up in routing loops. The TTL field puts an upper limit on these looping datagrams.

When a router gets an IP datagram whose TTL is either 0 or 1 it must not forward the datagram. (A destination host that receives a datagram like this can deliver it to the application, since the datagram does not have to be routed. Normally, however, no system should receive a datagram with a TTL of 0.) Instead the router throws away the datagram *and* sends back to the originating host an ICMP "time exceeded" message. The key to Traceroute is that the IP datagram containing this ICMP message has the router's IP address as the source address.

We can now guess the operation of Traceroute. It sends an IP datagram with a TTL of 1 to the destination host. The first router to handle the datagram decrements the TTL, discards the datagram, and sends back the ICMP time exceeded. This identifies the first router in the path. Traceroute then sends a datagram with a TTL of 2, and we find the IP address of the second router. This continues until the datagram reaches the destination host. But even though the arriving IP datagram has a TTL of 1, the destination host won't throw it away and generate the ICMP time exceeded, since the datagram has reached its final destination. How can we determine when we've reached the destination?

Traceroute sends UDP datagrams to the destination host, but it chooses the destination UDP port number to be an unlikely value (larger than 30,000), making it improbable that an application at the destination is using that port. This causes the destination host's UDP module to generate an ICMP "port unreachable" error (Section 6.5) when the datagram arrives. All Traceroute needs to do is differentiate between the received ICMP messages—time exceeded versus port unreachable—to know when it's done.

The Traceroute program must be able to set the TTL field in the outgoing datagram. Not all programming interfaces to TCP/IP support this, and not all implementations support the capability, but most current systems do, and are able to run Traceroute. This programming interface normally requires the user to have superuser privilege, meaning it may take special privilege to run it on your host.

8.3 LAN Output

We're now ready to run `traceroute` and see the output. We'll use our simple internet (see the figure on the inside front cover) going from `svr4` to `slip`, through the router `bsdi`. The hardwired SLIP link between `bsdi` and `slip` is 9600 bits/sec.

```
svr4 % traceroute slip
traceroute to slip (140.252.13.65), 30 hops max, 40 byte packets
 1  bsdi (140.252.13.35)  20 ms  10 ms  10 ms
 2  slip (140.252.13.65)  120 ms  120 ms  120 ms
```

The first unnumbered line of output gives the name and IP address of the destination and indicates that `traceroute` won't increase the TTL beyond 30. The datagram size of 40 bytes allows for the 20-byte IP header, the 8-byte UDP header, and 12 bytes of user data. (The 12 bytes of user data contain a sequence number that is incremented each time a datagram is sent, a copy of the outgoing TTL, and the time at which the datagram was sent.)

The next two lines in the output begin with the TTL, followed by the name of the host or router, and its IP address. For each TTL value three datagrams are sent. For each returned ICMP message the round-trip time is calculated and printed. If no response is received within 5 seconds for any of the three datagrams, an asterisk is printed instead and the next datagram is sent. In this output the first three datagrams had a TTL of 1 and the ICMP messages were returned in 20, 10, and 10 ms. The next three datagrams were sent with a TTL of 2 and the ICMP messages were returned 120 ms later. Since the TTL of 2 reached the final destination, the program then stopped.

The round-trip times are calculated by the `traceroute` program on the sending host. They are the total RTTs from the `traceroute` program to that router. If we're interested in the per-hop time we have to subtract the value printed for TTL N from the value printed for TTL $N+1$.

Figure 8.1 shows the `tcpdump` output for this run. As we might have guessed, the reason that the first probe packet to `bsdi` had an RTT of 20 ms and the next two had an RTT of 10 ms was because of an ARP exchange. `tcpdump` shows this is indeed the case.

The destination UDP port starts at 33435 and is incremented by one each time a datagram is sent. This starting port number can be changed with a command-line option. The UDP datagram contains 12 bytes of user data, which we calculated earlier when `traceroute` output that it was sending 40-byte datagrams.

Next, `tcpdump` prints the comment `[ttl 1]` when the IP datagram has a TTL of 1. It prints a message like this when the TTL is 0 or 1, to warn us that something looks funny in the datagram. Here we expect to see the TTL of 1, but with some other application it could be a warning that the datagram might not get to its final destination. We should never see a datagram passing by with a TTL of 0, unless the router that put it on the wire is broken.

```
 1  0.0                  arp who-has bsdi tell svr4
 2  0.000586 (0.0006)    arp reply bsdi is-at 0:0:c0:6f:2d:40

 3  0.003067 (0.0025)    svr4.42804 > slip.33435: udp 12 [ttl 1]
 4  0.004325 (0.0013)    bsdi > svr4: icmp: time exceeded in-transit

 5  0.069810 (0.0655)    svr4.42804 > slip.33436: udp 12 [ttl 1]
 6  0.071149 (0.0013)    bsdi > svr4: icmp: time exceeded in-transit

 7  0.085162 (0.0140)    svr4.42804 > slip.33437: udp 12 [ttl 1]
 8  0.086375 (0.0012)    bsdi > svr4: icmp: time exceeded in-transit

 9  0.118608 (0.0322)    svr4.42804 > slip.33438: udp 12
10  0.226464 (0.1079)    slip > svr4: icmp: slip udp port 33438 unreachable

11  0.287296 (0.0608)    svr4.42804 > slip.33439: udp 12
12  0.395230 (0.1079)    slip > svr4: icmp: slip udp port 33439 unreachable

13  0.409504 (0.0143)    svr4.42804 > slip.33440: udp 12
14  0.517430 (0.1079)    slip > svr4: icmp: slip udp port 33440 unreachable
```

Figure 8.1 `tcpdump` output for `traceroute` example from `svr4` to `slip`.

The ICMP message "time exceeded in transit" is what we expect to see from the router `bsdi`, since it will decrement the TTL to 0. The ICMP message comes from the router even though the IP datagram that was thrown away was going to `slip`.

There are two different ICMP "time exceeded" messages (Figure 6.3, p. 71), each with a different *code* field in the ICMP message. Figure 8.2 shows the format of this ICMP error message.

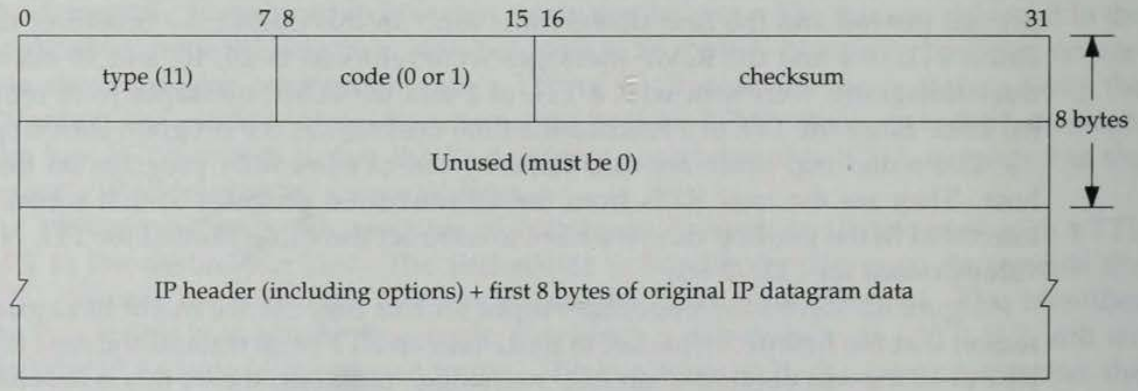


Figure 8.2 ICMP time exceeded message.

The one we've been describing is generated when the TTL reaches 0, and is specified by a *code* of 0.

It's also possible for a host to send an ICMP "time exceeded during reassembly" when it times out during the reassembly of a fragmented datagram. (We talk about fragmentation and reassembly in Section 11.5.) This error is specified by a *code* of 1.

Lines 9–14 in Figure 8.1 correspond to the three datagrams sent with a TTL of 2. These reach the final destination and generate an ICMP port unreachable message.

It is worthwhile to calculate what the round-trip times should be for the SLIP link, similar to what we did in Section 7.2 when we set the link to 1200 bits/sec for the Ping example. The outgoing UDP datagram contains 12 bytes of data, 8 bytes of UDP header, 20 bytes of IP header, and 2 bytes (at least) of SLIP framing (Section 2.4) for a total of 42 bytes. Unlike Ping, however, the size of the return datagrams changes. Recall from Figure 6.9 (p. 78) that the returned ICMP message contains the IP header of the datagram that caused the error and the first 8 bytes of data that followed that IP header (which is a UDP header in the case of `traceroute`). This gives us a total of 20 + 8 + 20 + 8 + 2, or 58 bytes. With a data rate of 960 bytes/sec the expected RTT is (42 + 58/960) or 104 ms. This corresponds to the 110-ms value measured on `svr4`.

The source port number in Figure 8.1 (42804) seems high. `traceroute` sets the source port number of the UDP datagrams that it sends to the logical-OR of its Unix process ID with 32768. In case `traceroute` is being run multiple times on the same host, each process looks at the source port number in the UDP header that's returned by ICMP, and only handles those messages that are replies to probes that it sent.

There are several points to note with `traceroute`. First, there is no guarantee that the route today will be in use tomorrow, or even that two consecutive IP datagrams follow the same route. If a route changes while the program is running you'll see it occur because `traceroute` prints the new IP address for the given TTL if it changes.

Second, there is no guarantee that the path taken by the returned ICMP message retraces the path of the UDP datagram sent by `traceroute`. This implies that the round-trip times printed may not be a true indication of the outgoing and returning datagram times. (If it takes 1 second for the UDP datagram to travel from the source to a router, but 3 seconds for the ICMP message to travel a different path back to the source, the printed round-trip time is 4 seconds.)

Third, the source IP address in the returned ICMP message is the IP address of the interface on the router on which the UDP datagram *arrived*. This differs from the IP record route option (Section 7.3), where the IP address recorded was the outgoing interface's address. Since every router by definition has two or more interfaces, running `traceroute` from host A to host B can generate different output than from host B to host A. Indeed, if we run `traceroute` from host `slip` to `svr4` the output becomes:

```
slip % traceroute svr4
traceroute to svr4 (140.252.13.34), 30 hops max, 40 byte packets
 1  bsdi (140.252.13.66)  110 ms  110 ms  110 ms
 2  svr4 (140.252.13.34)  110 ms  120 ms  110 ms
```

This time the IP address printed for host `bsdi` is 140.252.13.66, the SLIP interface, while previously it was 140.252.13.35, the Ethernet interface. Since `traceroute` also tries to print the name associated with an IP address, the names can change. (In our example both interfaces on `bsdi` have the same name.)

Consider Figure 8.3. It shows two local area networks with a router connected to each LAN. The two routers are connected with a point-to-point link. If we run `traceroute` from a host on the left LAN to a host on the right LAN, the IP addresses found for the routers will be *if1* and *if3*. But going the other way will print the IP addresses *if4* and *if2*. The two interfaces *if2* and *if3* share the same network ID, while the other two interfaces have different network IDs.

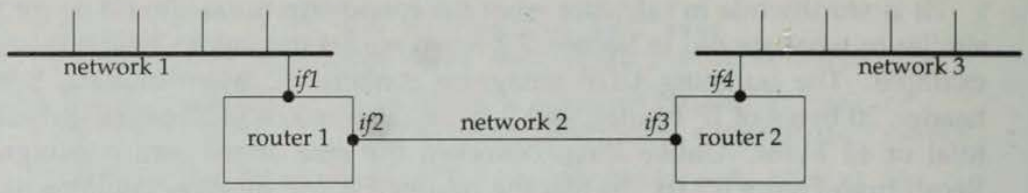


Figure 8.3 Identification of interfaces printed by `traceroute`.

Finally, across wide area networks the `traceroute` output is much easier to comprehend if the IP addresses are printed as readable domain names, instead of as IP addresses. But since the only piece of information `traceroute` has when it receives the ICMP message is an IP address, it does a "reverse name lookup" to find the name, given the IP address. This requires the administrator responsible for that router or host to configure their reverse name lookup function correctly (which isn't always the case). We describe how an IP address is converted to a name using the DNS in Section 14.5.

8.4 WAN Output

The output shown earlier for our small internet is adequate for examining the protocols in action, but more a realistic use of `traceroute` involves larger internets such as the worldwide Internet.

Figure 8.4 is from the host `sun` to the Network Information Center, the NIC.

```
sun % traceroute nic.ddn.mil
traceroute to nic.ddn.mil (192.112.36.5), 30 hops max, 40 byte packets
 1  netb.tuc.noao.edu (140.252.1.183)  218 ms  227 ms  233 ms
 2  gateway.tuc.noao.edu (140.252.1.4)  233 ms  229 ms  204 ms
 3  butch.telcom.arizona.edu (140.252.104.2)  204 ms  228 ms  234 ms
 4  Gabby.Telcom.Arizona.EDU (128.196.128.1)  234 ms  228 ms  204 ms
 5  NSIgate.Telcom.Arizona.EDU (192.80.43.3)  233 ms  228 ms  234 ms
 6  JPL1.NSN.NASA.GOV (128.161.88.2)  234 ms  590 ms  262 ms
 7  JPL3.NSN.NASA.GOV (192.100.15.3)  238 ms  223 ms  234 ms
 8  GSFC3.NSN.NASA.GOV (128.161.3.33)  293 ms  318 ms  324 ms
 9  GSFC8.NSN.NASA.GOV (192.100.13.8)  294 ms  318 ms  294 ms
10  SURA2.NSN.NASA.GOV (128.161.166.2)  323 ms  319 ms  294 ms
11  nsn-FIX-pe.sura.net (192.80.214.253)  294 ms  318 ms  294 ms
12  GSI.NSN.NASA.GOV (128.161.252.2)  293 ms  318 ms  324 ms
13  NIC.DDN.MIL (192.112.36.5)  324 ms  321 ms  324 ms
```

Figure 8.4 `traceroute` from host `sun` to `nic.ddn.mil`.

Since running this example for inclusion in the text, the NIC for non-DDN sites (i.e., non-military) has moved from `nic.ddn.mil` to `rs.internic.net`, the new "InterNIC."

Once the datagrams leave the `tuc.noao.edu` network they enter the `telcom.arizona.edu` network. They then enter the NASA Science Internet, `nsn.nasa.gov`. The routers for TTLs 6 and 7 are at the Jet Propulsion Laboratory (JPL). The network `sura.net` in the output for TTL 11 is the Southeastern Universities Research Association Network. The name GSI at TTL 12 is Government Systems, Inc., the operator of the NIC.

The second RTT for the TTL of 6 (590) is almost double the other two RTTs (234 and 262). This illustrates the dynamics of IP routing. Something happened somewhere between the sending host and this router that slowed down this datagram. Also, we can't tell if it was the outbound datagram that got held up or the return ICMP error.

The RTT for the first probe with a TTL of 3 (204) is less than the RTT for the first probe with a TTL of 2 (233). Since each printed RTT is the total time from the sending host to that router, this can (and does) happen.

The example in Figure 8.5 is from the host `sun` to the author's publisher.

```
sun % traceroute aw.com
traceroute to aw.com (192.207.117.2), 30 hops max, 40 byte packets

 1  netb.tuc.noao.edu (140.252.1.183)  227 ms  227 ms  234 ms
 2  gateway.tuc.noao.edu (140.252.1.4)  233 ms  229 ms  234 ms

 3  butch.telcom.arizona.edu (140.252.104.2)  233 ms  229 ms  234 ms
 4  Gabby.Telcom.Arizona.EDU (128.196.128.1)  264 ms  228 ms  234 ms
 5  Westgate.Telcom.Arizona.EDU (192.80.43.2)  234 ms  228 ms  234 ms

 6  uu-ua.AZ.westnet.net (192.31.39.233)  263 ms  258 ms  264 ms
 7  enss142.UT.westnet.net (192.31.39.21)  263 ms  258 ms  264 ms

 8  t3-2.Denver-cnss97.t3.ans.net (140.222.97.3)  293 ms  288 ms  275 ms
 9  t3-3.Denver-cnss96.t3.ans.net (140.222.96.4)  283 ms  263 ms  261 ms
10  t3-1.St-Louis-cnss80.t3.ans.net (140.222.80.2)  282 ms  288 ms  294 ms
11  t3-1.Chicago-cnss24.t3.ans.net (140.222.24.2)  293 ms  288 ms  294 ms
12  t3-2.Cleveland-cnss40.t3.ans.net (140.222.40.3)  294 ms  288 ms  294 ms
13  t3-1.New-York-cnss32.t3.ans.net (140.222.32.2)  323 ms  318 ms  324 ms
14  t3-1.Washington-DC-cnss56.t3.ans.net (140.222.56.2)  323 ms  318 ms  324 ms
15  t3-0.Washington-DC-cnss58.t3.ans.net (140.222.58.1)  324 ms  318 ms  324 ms
16  t3-0.enss136.t3.ans.net (140.222.136.1)  323 ms  318 ms  324 ms

17  Washington.DC.ALTER.NET (192.41.177.248)  323 ms  377 ms  324 ms
18  Boston.MA.ALTER.NET (137.39.12.2)  324 ms  347 ms  324 ms
19  AW-gw.ALTER.NET (137.39.62.2)  353 ms  378 ms  354 ms

20  aw.com (192.207.117.2)  354 ms  349 ms  354 ms
```

Figure 8.5 `traceroute` from host `sun.tuc.noao.edu` to `aw.com`.

This time the datagrams enter the regional network `westnet.net` (TTLs 6 and 7) after leaving the `telcom.arizona.edu` network. They then enter the NSFNET backbone, `t3.ans.net`, which is run by Advanced Network & Services. (T3 is the common abbreviation for the 45 Mbits/sec phone lines used by the backbone.) The final network is `alter.net`, the connection point to the Internet for `aw.com`.

8.5 IP Source Routing Option

Normally IP routing is dynamic with each router making a decision about which next-hop router to send the datagram to. Applications have no control of this, and are normally not concerned with it. It takes tools such as Traceroute to figure out what the route really is.

The idea behind source routing is that the sender specifies the route. Two forms are provided:

- *Strict* source routing. The sender specifies the *exact* path that the IP datagram must follow. If a router encounters a next hop in the source route that isn't on a directly connected network, an ICMP "source route failed" error is returned.
- *Loose* source routing. The sender specifies a list of IP address that the datagram must traverse, but the datagram can also pass through other routers between any two addresses in the list.

Traceroute provides a way to look at source routing, as we can specify an option allowing us to force a source route, and see what happens.

> Some of the publicly available Traceroute source code packages contain patches to specify loose source routing. But the standard versions normally don't include this option. A comment in the patches is that "Van Jacobson's original traceroute (spring 1988) supported this feature, but he removed it due to pressure from people with broken gateways." For the examples shown in this section, the author installed these patches and modified them to allow both loose and strict source routing.

Figure 8.6 shows the format of the source route option.

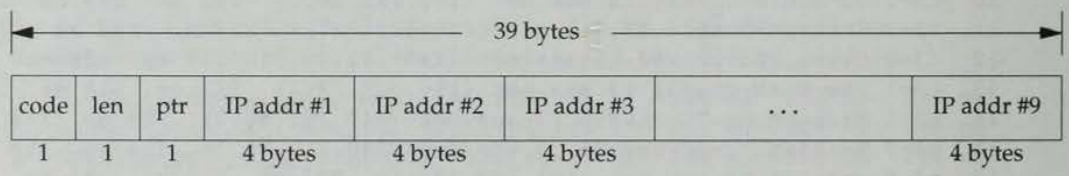


Figure 8.6 General format of the source route option in the IP header.

This format is nearly identical to the format of the record route option that we showed in Figure 7.3. But with source routing we have to fill in the list of IP addresses before sending the IP datagram, while with the record route option we allocate room and zero out the list of IP addresses, letting the routers fill in the next entry in the list. Also, with source routing we only allocate room for and initialize the number of IP addresses required, normally fewer than nine. With the record route option we allocated as much room as we could, for up to nine addresses.

The *code* is `0x83` for loose source routing, and `0x89` for strict source routing. The *len* and *ptr* fields are identical to what we described in Section 7.3.

The source route options are actually called "source and record route" (LSRR and SSRR, for loose and strict) since the list of IP addresses is updated as the datagram passes along the path. What happens is as follows:

- The sending host takes the source route list from the application, removes the first entry (it becomes the destination address of the datagram), moves all the remaining entries left by one entry (where left is as in Figure 8.6), and places the original destination address as the final entry in the list. The pointer still points to the first entry in the list (e.g., the value of the pointer is 4).
- Each router that handles the datagram checks whether it is the destination address of the datagram. If not, the datagram is forwarded as normal. (In this case loose source routing must have been specified, or we wouldn't have received the datagram.)
- If the router is the destination, and the pointer is not greater than the length, then (1) the next address in the list (where *ptr* points) becomes the destination address of the datagram, (2) the IP address corresponding to the outgoing interface replaces the source address just used, and (3) the pointer is incremented by 4.

This is best explained with an example. In Figure 8.7 we assume that the sending application on host *S* sends a datagram to *D*, specifying a source route of *R1*, *R2*, and *R3*.

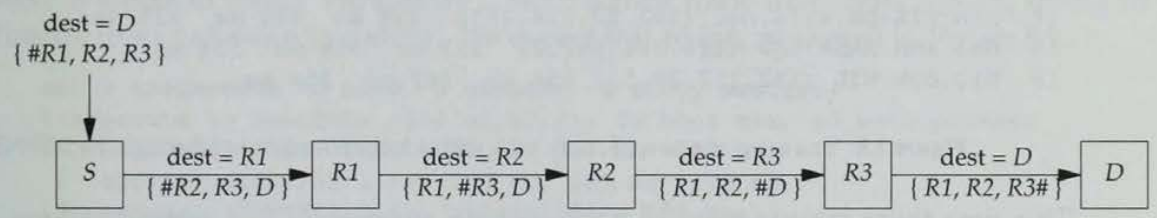


Figure 8.7 Example of IP source routing.

In this figure the pound sign (#) denotes the pointer field, which assumes the values of 4, 8, 12, and 16. The length field will always be 15 (three IP addresses plus 3 bytes of overhead). Notice how the destination address of the IP datagram changes on every hop.

When an application receives data that was source routed, it should fetch the value of the received route and supply a reversed route for sending replies.

> The Host Requirements RFC specifies that a TCP client must be able to specify a source route, and that a TCP server must be able to receive a source route, and use the reverse route for all segments on that TCP connection. If the TCP server later receives a different source route, that newer source route overrides the earlier one.

`traceroute` Examples with Loose Source Routing

The `-g` option to `traceroute` lets us specify intermediate routers to be used with loose source routing. This option can be specified up to eight times. (The reason this is eight and not nine is that the programming interface being used requires that the final entry be the destination.)

Recall from Figure 8.4 that the route to the NIC, `nic.ddn.mil`, was through the NASA Science Internet. In Figure 8.8 we force the datagrams to pass through the NSFNET instead by specifying the router `enss142.UT.westnet.net` (192.31.39.21) as an intermediate router:

```
sun % traceroute -g 192.31.39.21  nic.ddn.mil
traceroute to nic.ddn.mil (192.112.36.5), 30 hops max, 40 byte packets
 1  netb.tuc.noao.edu (140.252.1.183)  259 ms  256 ms  235 ms

 2  butch.telcom.arizona.edu (140.252.104.2)  234 ms  228 ms  234 ms
 3  Gabby.Telcom.Arizona.EDU (128.196.128.1)  234 ms  257 ms  233 ms

 4  enss142.UT.westnet.net (192.31.39.21)  294 ms  288 ms  295 ms

 5  t3-2.Denver-cnss97.t3.ans.net (140.222.97.3)  294 ms  286 ms  293 ms
 6  t3-3.Denver-cnss96.t3.ans.net (140.222.96.4)  293 ms  288 ms  294 ms
 7  t3-1.St-Louis-cnss80.t3.ans.net (140.222.80.2)  294 ms  318 ms  294 ms
 8  * t3-1.Chicago-cnss24.t3.ans.net (140.222.24.2)  318 ms  295 ms
 9  t3-2.Cleveland-cnss40.t3.ans.net (140.222.40.3)  319 ms  318 ms  324 ms
10  t3-1.New-York-cnss32.t3.ans.net (140.222.32.2)  324 ms  318 ms  324 ms
11  t3-1.Washington-DC-cnss56.t3.ans.net (140.222.56.2)  353 ms  348 ms  325 ms
12  t3-0.Washington-DC-cnss58.t3.ans.net (140.222.58.1)  348 ms  347 ms  325 ms
13  t3-0.enss145.t3.ans.net (140.222.145.1)  353 ms  348 ms  325 ms

14  nsn-FIX-pe.sura.net (192.80.214.253)  353 ms  348 ms  325 ms
15  GSI.NSN.NASA.GOV (128.161.252.2)  353 ms  348 ms  354 ms
16  NIC.DDN.MIL (192.112.36.5)  354 ms  347 ms  354 ms
```

Figure 8.8 `traceroute` to `nic.ddn.mil` with a loose source route through the NSFNET.

This time there appear to be 16 hops with an average RTT of around 350 ms, while the normal route shown in Figure 8.4 had only 13 hops and an average RTT of around 322 ms. The default route appears better. (There are also other decisions made when routes are established. Some are made on the basis of the organizational and political boundaries of the networks involved.)

But we said there *appear to be* 16 hops, because a comparison of this output with our previous example through the NSFNET (Figure 8.5) shows three missing routers in this example using loose source routing. (These are probably caused by bugs in the router's generation of ICMP time exceeded errors in response to source routed datagrams.) The router `gateway.tuc.noao.edu` is missing between `netb` and `butch`, and the routers `Westgate.Telcom.Arizona.edu` and `uu-ua.AZ.westnet.net` are also missing between `Gabby` and `enss142.UT.westnet.net`. There is probably a software problem in these missing routers related to incoming datagrams with the loose source routing option. There are really 19 hops between the source and the NIC, when using the NSFNET. Exercise 8.5 continues the discussion of these missing routers.

This example also illustrates another problem. On the command line we have to specify the dotted-decimal IP address of the router `enss142.UT.westnet.net` instead of its name. This is because the reverse name lookup (return the name, given the IP address, Section 14.5), associates the name with the IP address, but the forward lookup (given the name, return the IP address) fails. The forward mapping and reverse mapping are two separate files in the DNS (Domain Name System) and not all

administrators keep the two synchronized with each other. It's not uncommon to have one direction work and the other direction fail.

Something that we haven't seen before is the asterisk (*) printed for the first RTT for the TTL of 8. This indicates that a timeout occurred and no response was received within 5 seconds for this probe.

Another point that we can infer from a comparison of this figure and Figure 8.4 is that the router `nsn-FIX-pe.sura.net` is connected to both the NSFNET and the NASA Science Internet.

`traceroute` Examples with Strict Source Routing

The `-G` option in the author's version of `traceroute` is identical to the `-g` option described earlier, but the source route is strict instead of loose. We can use this to see what happens when an invalid strict source route is specified. Recall from Figure 8.5 that the normal sequence of routers for datagrams from the author's subnet to the NSFNET is through `netb`, `gateway`, `butch`, and `gabby`. (We've omitted the domain suffixes, `.tuc.noao.edu` and `.telcom.arizona.edu`, in all the output below to make it easier to read.) We specify a strict source route that omits `butch`, trying to go directly from `gateway` to `gabby`. We expect this to fail, as shown in Figure 8.9.

```
sun % traceroute -G netb -G gateway -G gabby westgate
traceroute to westgate (192.80.43.2), 30 hops max, 40 byte packets
 1  netb (140.252.1.183)  272 ms  257 ms  261 ms
 2  gateway (140.252.1.4)  263 ms  259 ms  234 ms
 3  gateway (140.252.1.4)  263 ms !S *  235 ms !S
```

Figure 8.9 `traceroute` with a strict source route that fails.

The key here is the notation `!S` following the RTTs for the TTL of 3. This indicates that `traceroute` received an ICMP "source route failed" error message: a *type* of 3 and a *code* of 5 from Figure 6.3. The asterisk for the second RTT for the TTL of 3 indicates no response was received for that probe. This is what we expect, since it's impossible for `gateway` to send the datagram directly to `gabby`, because they're not directly connected.

The reason that both TTLs 2 and 3 are from `gateway` is that the values for the TTL of 2 are from `gateway` when it receives the datagram with an incoming TTL of 1. It discovers that the TTL has expired before it looks at the (invalid) strict source route, and sends back the ICMP time exceeded. The line with a TTL of 3 is received by `gateway` with an incoming TTL of 2, so it looks at the strict source route, discovers that it's invalid, and sends back the ICMP source route failed error.

Figure 8.10 shows the `tcpdump` output corresponding to this example. This output was collected on the SLIP link between `sun` and `netb`. We had to specify the `-v` option for `tcpdump` to display the source route information. This produces other output that we don't need, such as the datagram ID, which we've deleted. Also, the notation `SSRR` stands for "strict source and record route."

```
 1  0.0                  sun.33593 > netb.33435: udp 12 [ttl 1]
                         (optlen=16 SSRR{#gateway gabby westgate} EOL)
 2  0.270278 (0.2703)    netb > sun: icmp: time exceeded in-transit
 3  0.284784 (0.0145)    sun.33593 > netb.33436: udp 12 [ttl 1]
                         (optlen=16 SSRR{#gateway gabby westgate} EOL)
 4  0.540338 (0.2556)    netb > sun: icmp: time exceeded in-transit
 5  0.550062 (0.0097)    sun.33593 > netb.33437: udp 12 [ttl 1]
                         (optlen=16 SSRR{#gateway gabby westgate} EOL)
 6  0.810310 (0.2602)    netb > sun: icmp: time exceeded in-transit
 7  0.818030 (0.0077)    sun.33593 > netb.33438: udp 12 (ttl 2,
                         optlen=16 SSRR{#gateway gabby westgate} EOL)
 8  1.080337 (0.2623)    gateway > sun: icmp: time exceeded in-transit
 9  1.092564 (0.0122)    sun.33593 > netb.33439: udp 12 (ttl 2,
                         optlen=16 SSRR{#gateway gabby westgate} EOL)
10  1.350322 (0.2578)    gateway > sun: icmp: time exceeded in-transit
11  1.357382 (0.0071)    sun.33593 > netb.33440: udp 12 (ttl 2,
                         optlen=16 SSRR{#gateway gabby westgate} EOL)
12  1.590586 (0.2332)    gateway > sun: icmp: time exceeded in-transit
13  1.598926 (0.0083)    sun.33593 > netb.33441: udp 12 (ttl 3,
                         optlen=16 SSRR{#gateway gabby westgate} EOL)
14  1.860341 (0.2614)    gateway > sun:
                         icmp: gateway unreachable - source route failed
15  1.875230 (0.0149)    sun.33593 > netb.33442: udp 12 (ttl 3,
                         optlen=16 SSRR{#gateway gabby westgate} EOL)
16  6.876579 (5.0013)    sun.33593 > netb.33443: udp 12 (ttl 3,
                         optlen=16 SSRR{#gateway gabby westgate} EOL)
17  7.110518 (0.2339)    gateway > sun:
                         icmp: gateway unreachable - source route failed
```

Figure 8.10 `tcpdump` output of `traceroute` with failed strict source route.

First note that each UDP datagram sent by `sun` has a destination of `netb`, not the destination host (`westgate`). We described this with the example shown in Figure 8.7. Similarly, the other two routers specified with the `-G` option (`gateway` and `gabby`) and the final destination (`westgate`) become the SSRR option list on the first hop.

We can also see from this output that the timeout used by `traceroute` (the time difference between lines 15 and 16) is 5 seconds.

`traceroute` Round Trips with Loose Source Routing

Earlier we said that there is no guarantee that the route from A to B is the same as the route from B to A. Other than having a login on both systems and running `traceroute` on each end, it's hard to find out if there is a difference in the two paths. Using loose source routing, however, we can determine the route in both directions.

The trick is to specify loose source routing with the destination as the loose route, and the sending host as the final destination. For example, on the host `sun` we can find the paths to and from the host `bruno.cs.colorado.edu` (Figure 8.11) .

```
sun % traceroute -g bruno.cs.colorado.edu sun
traceroute to sun (140.252.13.33), 30 hops max, 40 byte packets
 1  netb.tuc.noao.edu (140.252.1.183)  230 ms  227 ms  233 ms
 2  gateway.tuc.noao.edu (140.252.1.4)  233 ms  229 ms  234 ms
 3  butch.telcom.arizona.edu (140.252.104.2)  234 ms  229 ms  234 ms
 4  Gabby.Telcom.Arizona.EDU (128.196.128.1)  233 ms  231 ms  234 ms
 5  NSIgate.Telcom.Arizona.EDU (192.80.43.3)  294 ms  258 ms  234 ms
 6  JPL1.NSN.NASA.GOV (128.161.88.2)  264 ms  258 ms  264 ms
 7  JPL2.NSN.NASA.GOV (192.100.15.2)  264 ms  258 ms  264 ms
 8  NCAR.NSN.NASA.GOV (128.161.97.2)  324 ms *  295 ms
 9  cu-gw.ucar.edu (192.43.244.4)  294 ms  318 ms  294 ms
10  engr-gw.Colorado.EDU (128.138.1.3)  294 ms  288 ms  294 ms
11  bruno.cs.colorado.edu (128.138.243.151)  293 ms  317 ms  294 ms
12  engr-gw-ot.cs.colorado.edu (128.138.204.1)  323 ms  317 ms  384 ms
13  cu-gw.Colorado.EDU (128.138.1.1)  294 ms  318 ms  294 ms
14  enss.ucar.edu (192.43.244.10)  323 ms  318 ms  294 ms
15  t3-1.Denver-cnss97.t3.ans.net (140.222.97.2)  294 ms  288 ms  384 ms
16  t3-0.enss142.t3.ans.net (140.222.142.1)  293 ms  288 ms  294 ms
17  Gabby.Telcom.Arizona.EDU (192.80.43.1)  294 ms  288 ms  294 ms
18  Butch.Telcom.Arizona.EDU (128.196.128.88)  293 ms  317 ms  294 ms
19  gateway.tuc.noao.edu (140.252.104.1)  294 ms  289 ms  294 ms
20  netb.tuc.noao.edu (140.252.1.183)  324 ms  321 ms  294 ms
21  sun.tuc.noao.edu (140.252.13.33)  534 ms  529 ms  564 ms
```

Figure 8.11 `traceroute` example showing unsymmetrical routing path.

The outbound path (TTLs 1–11) differs from the return path (TTLs 11–21), a good illustration that Internet routing need not be symmetrical.

This output also illustrates the point we discussed with Figure 8.3. Compare the output for TTLs 2 and 19: both are for the router `gateway.tuc.noao.edu`, but the two IP addresses are different. Since `traceroute` identifies the incoming interface, and since we're passing through the router in two different directions, once on the outbound path (TTL 2) and then on the return path (TTL 19), we expect this. We see the same effect comparing TTLs 3 and 18, and TTLs 4 and 17.

8.6 Summary

Traceroute is an indispensable tool when working with a TCP/IP network. Its operation is simple: send UDP datagrams starting with a TTL of 1, increasing the TTL by 1, to locate each router in the path. An ICMP time exceeded is returned by each router when it discards the UDP datagram, and an ICMP port unreachable is generated by the final destination.

We ran examples of `traceroute` on both LANs and WANs, and used it to examine IP source routing. We used loose source routing to see if the route to a destination is the same as the return route from that destination.

Exercises

8.1 What can happen if an IP implementation decrements the incoming TTL and then tests for equal to 0?

8.2 How does `traceroute` calculate the RTT? Compare this to the RTT calculation done by `ping`.

8.3 (This exercise and the next one are based on actual problems determined when `traceroute` was being developed, and are from comments in the `traceroute` source code.) Assume there are three routers (R1, R2, and R3) between the source and destination and that the middle router (R2) decrements the TTL but incorrectly forwards the IP datagram when the incoming TTL was 1. Describe what happens. How can you see this occur when running `traceroute`?

8.4 Again assume there are three routers between the source and destination. This time the destination host has a bug whereby it always uses the incoming TTL as the outgoing TTL of an ICMP message. Describe what happens and how you would see this.

8.5 We can run `tcpdump` on the SLIP link between `sun` and `netb` when running the example from Figure 8.8. If we specify the `-v` option we can see the TTL value of the returned ICMP messages. Doing this shows the incoming TTL from `netb` to be 255, from `butch` it's 253, from `Gabby` it's 252, and from `enss142.UT.westnet.net` it's 249. Does this give any additional information about whether there really are some missing routers?

8.6 Both SunOS and SVR4 provide a version of `ping` with a `-l` option that provides a loose source route. The manual pages state that it's intended to be used with the `-R` option (which specifies the record route option). If you have access to either of these systems, try these two options together. What's happening? If you can watch the datagrams with `tcpdump`, describe what's going on.

8.7 Compare the ways `ping` and `traceroute` handle multiple instances of the client on the same host.

8.8 Compare the ways `ping` and `traceroute` measure the round-trip time.

8.9 We said `traceroute` picks the starting UDP destination port number at 33435 and increments this by one for each packet sent. In Section 1.9 we said ephemeral port numbers are normally between 1024 and 5000, making it unlikely that Traceroute's destination port is in use on the destination host. Is this still true under Solaris 2.2? (*Hint*: Read Section E.4.)

8.10 Read RFC 1393 [Malkin 1993b] for a proposed alternative way of determining the path to a destination. What are its advantages and disadvantages?

9

IP Routing

9.1 Introduction

Routing is one of the most important functions of IP. Figure 9.1 shows a simplified view of the processing done at the IP layer. Datagrams to be routed can be generated either on the local host or on some other host. In the latter case this host must be configured to act as a router, or datagrams received through the network interfaces that are not ours are dropped (i.e., silently discarded).

In Figure 9.1 we also show a routing daemon, which is normally a user process. The most common daemons used on Unix systems are the programs `routed` and `gated`. (The term *daemon* means the process is running "in the background," carrying out operations on behalf of the whole system. Daemons are normally started when the system is bootstrapped and run as long as the system is up.) The topics of which routing protocol to use on a given host, how to exchange routing information with adjacent routers, and how the routing protocols work are complex and can fill an entire book of their own. (Interested readers are referred to [Perlman 1992] for many of the details.) We'll look briefly at dynamic routing and the Routing Information Protocol (RIP) in Chapter 10. Our main interest in the current chapter is how a single IP layer makes its routing decisions.

The routing table that we show in Figure 9.1 is accessed frequently by IP (on a busy host this could mean hundreds of times a second) but is updated much less frequently by a routing daemon (possibly about once every 30 seconds). The routing table can also be updated when ICMP "redirect" messages are received, something we'll look at in Section 9.5, and by the `route` command. This command is often executed when the system is bootstrapped, to install some initial routes. We'll also use the `netstat` command in this chapter to display the routing table.

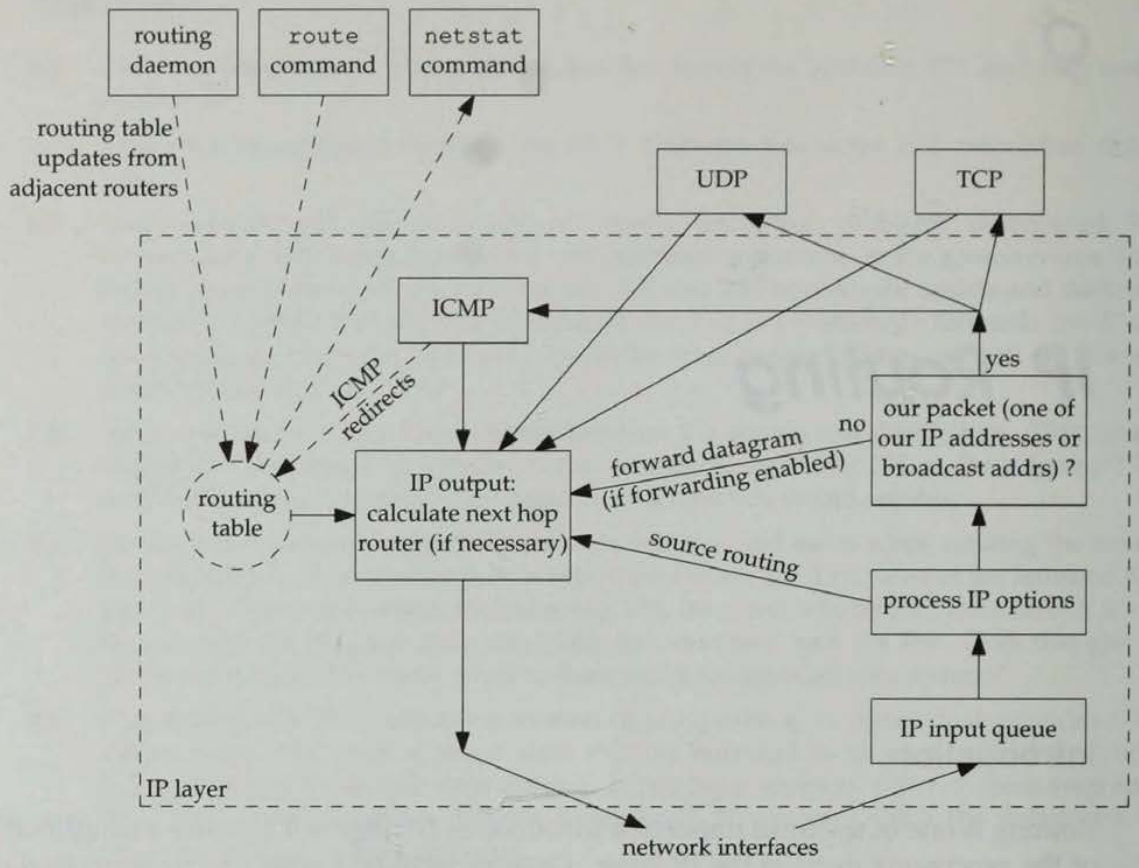


Figure 9.1 Processing done at the IP layer.

9.2 Routing Principles

The place to start our discussion of IP routing is to understand what is maintained by the kernel in its routing table. The information contained in the routing table drives all the routing decisions made by IP.

In Section 3.3 we listed the steps that IP performs when it searches its routing table.

1. Search for a matching host address.
2. Search for a matching network address.
3. Search for a default entry. (The default entry is normally specified in the routing table as a network entry, with a network ID of 0.)

A matching host address is always used before a matching network address.

The routing done by IP, when it searches the routing table and decides which interface to send a packet out, is a *routing mechanism*. This differs from a *routing policy*, which is a set of rules that decides which routes go into the routing table. IP performs the routing mechanism while a routing daemon normally provides the routing policy.

Simple Routing Table

Let's start by looking at some typical host routing tables. On the host `svr4` we execute the `netstat` command with the `-r` option to list the routing table and the `-n` option, which prints IP addresses in numeric format, rather than as names. (We do this because some of the entries in the routing table are for networks, not hosts. Without the `-n` option, the `netstat` command searches the file `/etc/networks` for the network names. This confuses the discussion by adding another set of names—network names in addition to hostnames.)

```
svr4 % netstat -rn
Routing tables
Destination       Gateway           Flags    Refcnt Use      Interface
140.252.13.65     140.252.13.35     UGH      0      0        emd0
127.0.0.1         127.0.0.1         UH       1      0        lo0
default           140.252.13.33     UG       0      0        emd0
140.252.13.32     140.252.13.34     U        4      25043    emd0
```

The first line says for destination 140.252.13.65 (host `slip`) the gateway (router) to send the packet to is 140.252.13.35 (`bsdi`). This is what we expect, since the host `slip` is connected to `bsdi` with a SLIP link, and `bsdi` is on the same Ethernet as this host.

There are five different flags that can be printed for a given route.

- `U` The route is up.
- `G` The route is to a gateway (router). If this flag is not set, the destination is directly connected.
- `H` The route is to a host, that is, the destination is a complete host address. If this flag is not set, the route is to a network, and the destination is a network address: a net ID, or a combination of a net ID and a subnet ID.
- `D` The route was created by a redirect (Section 9.5).
- `M` The route was modified by a redirect (Section 9.5).

The `G` flag is important because it differentiates between an *indirect route* and a *direct route*. (The `G` flag is not set for a direct route.) The difference is that a packet going out a direct route has both the IP address and the link-layer address specifying the destination (Figure 3.3, p. 40). When a packet is sent out an indirect route, the IP address specifies the final destination but the link-layer address specifies the gateway (that is, the next-hop router). We saw an example of this in Figure 3.4 (p. 41). In this routing table example we have an indirect route (the `G` flag is set) so the IP address of a packet using this route is the final destination (140.252.13.65), but the link-layer address must correspond to the router 140.252.13.35.

It's important to understand the difference between the `G` and `H` flags. The `G` flag differentiates between a direct and an indirect route, as described above. The `H` flag, however, specifies that the destination address (the first column of `netstat` output) is a complete host address. The absence of the `H` flag means the destination address is a network address (the host ID portion will be 0). When the routing table is searched for

a route to a destination IP address, a host address entry must match the destination address completely, while a network address only needs to match the network ID and any subnet ID of the destination address. Also, most versions of the `netstat` command print all the host entries first, followed by the network entries.

The reference count column gives the number of active uses for each route. A connection-oriented protocol such as TCP holds on to a route while the connection is established. If we established a Telnet connection between the two hosts `svr4` and `slip`, we would see the reference count go to 1. With another Telnet connection the reference count would go to 2, and so on.

The next column ("use") displays the number of packets sent through that route. If we are the only users of the route and we run the `ping` program to send 5 packets, the count goes up by 5. The final column, the interface, is the name of the local interface.

The second line of output is for the loopback interface (Section 2.7), always named `lo0`. The `G` flag is not set, since the route is not to a gateway. The `H` flag indicates that the destination address (127.0.0.1) is a host address, and not a network address. When the `G` field is not set, indicating a direct route, the gateway column gives the IP address of the outgoing interface.

The third line of output is for the default route. Every host can have one or more default routes. This entry says to send packets to the router 140.252.13.33 (`sun`) if a more specific route can't be found. This means the current host (`svr4`) can access other systems across the Internet through the router `sun` (and its SLIP link), using this single routing table entry. Being able to establish a default route is a powerful concept. The flags for this route (`UG`) indicate that it's a route to a gateway, as we expect.

> Here we purposely call `sun` a router and not a host because when it's used as a default router, its IP forwarding function is being used, not its host functionality.
>
> The Host Requirements RFC specifically states that the IP layer must support multiple default routes. Many implementations, however, don't support this. When multiple default routes exist, a common technique is to round robin among them. This is what Solaris 2.2 does, for example.

The final line of output is for the attached Ethernet. The `H` flag is not set, indicating that the destination address (140.252.13.32) is a network address with the host portion set to 0. Indeed, the low-order 5 bits are 0 (Figure 3.11, p. 47). Since this is a direct route (the `G` flag is not set) the gateway column specifies the IP address of the outgoing interface.

Implied in this final entry, but not shown by the `netstat` output, is the subnet mask associated with this destination address (140.252.13.32). If this destination is being compared against the IP address 140.252.13.33, the address is first logically ANDed with the subnet mask associated with the destination (`0xfffffe0`, from Section 3.7) before the comparison. Since the kernel knows the interface associated with each routing table entry, and since each interface has an associated subnet mask, each routing table entry has an implied subnet mask.

The complexity of a host's routing table depends on the topology of the networks to which the host has access.

1. The simplest (but least interesting) case is a host that is not connected to any networks at all. The TCP/IP protocols can still be used on the host, but only to communicate with itself! The routing table in this case consists of a single entry for the loopback interface.
2. Next is a host connected to a single LAN, only able to access hosts on that LAN. The routing table consists of two entries: one for the loopback interface and one for the LAN (such as an Ethernet).
3. The next step occurs when other networks (such as the Internet) are reachable through a single router. This is normally handled with a default entry pointing to that router.
4. The final step is when other host-specific or network-specific routes are added. In our example the route to the host `slip`, through the router `bsdi`, is an example of this.

Let's follow through the steps IP performs when using this routing table to route some example packets on the host `svr4`.

1. Assume the destination address is the host `sun`, 140.252.13.33. A search is first made for a matching host entry. The two host entries in the table (`slip` and `localhost`) don't match, so a search is made through the routing table again for a matching network address. A match is found with the entry 140.252.13.32 (the network IDs and subnet IDs match), so the `emd0` interface is used. This is a direct route, so the link-layer address will be the destination address.
2. Assume the destination address is the host `slip`, 140.252.13.65. The first search through the table, for a matching host address, finds a match. This is an indirect route so the destination IP address remains 140.252.13.65, but the link-layer address must be the link-layer address of the gateway 140.252.13.35, and the interface is `emd0`.
3. This time we're sending a datagram across the Internet to the host `aw.com` (192.207.117.2). The first search of the routing table for a matching host address fails, as does the second search for a matching network address. The final step is a search for a default entry, and this succeeds. The route is an indirect route through the gateway 140.252.13.33 using the interface `emd0`.
4. In our final example we send a datagram to our own host. There are four ways to do this, using either the hostname, the host IP address, the loopback name, or the loopback IP address:

```
ftp svr4
ftp 140.252.13.32

ftp localhost
ftp 127.0.0.1
```

In the first two cases, the second search of the routing table yields a network match with 140.252.13.32, and the packet is sent down to the Ethernet driver. As we showed in Figure 2.4 (p. 28) it will be seen that this packet is destined for the host's own IP address, and the packet is sent to the loopback driver, which sends it to the IP input queue.

In the latter two cases, specifying the name of the loopback interface or its IP address, the first search of the routing table finds the matching host address entry, and the packet is sent to the loopback driver, which sends it to the IP input queue.

In all four cases the packet is sent to the loopback driver, but two different routing decisions are made.

Initializing a Routing Table

We never said how these routing table entries are created. Whenever an interface is initialized (normally when the interface's address is set by the `ifconfig` command) a direct route is automatically created for that interface. For point-to-point links and the loopback interface, the route is to a host (i.e., the `H` flag is set). For broadcast interfaces such as an Ethernet, the route is to that network.

Routes to hosts or networks that are not directly connected must be entered into the routing table somehow. One common way is to execute the `route` command explicitly from the initialization files when the system is bootstrapped. On the host `svr4` the following two commands were executed to add the entries that we showed earlier:

```
route add default sun 1
route add slip bsdi 1
```

The third arguments (`default` and `slip`) are the destinations, the fourth argument is the gateway (router), and the final argument is a routing metric. All that the `route` command does with this metric is install the route with the `G` flag set if the metric is greater than 0, or without the `G` flag if the metric is 0.

> Unfortunately, few systems agree on which start-up file contains the `route` commands. Under 4.4BSD and BSD/386 it is `/etc/netstart`, under SVR4 it is `/etc/inet/rc.inet`, under Solaris 2.x it is `/etc/rc2.d/S69inet`, SunOS 4.1.x uses `/etc/rc.local`, and AIX 3.2.2 uses `/etc/rc.net`.

Some systems allow a default router to be specified in a file such as `/etc/defaultrouter`, and this default is added to the routing table on every reboot.

Other ways to initialize a routing table are to run a routing daemon (Chapter 10) or to use the newer router discovery protocol (Section 9.6).

A More Complex Routing Table

The host `sun` is the default router for all the hosts on our subnet, since it has the dialup SLIP link that connects to the Internet (see the figure on the inside front cover).

```
sun % netstat -rn
Routing tables
Destination       Gateway           Flags    Refcnt Use      Interface
140.252.13.65     140.252.13.35     UGH      0      171      le0
127.0.0.1         127.0.0.1         UH       1      766      lo0
140.252.1.183     140.252.1.29      UH       0      0        sl0
default           140.252.1.183     UG       1      2955     sl0
140.252.13.32     140.252.13.33     U        8      99551    le0
```

The first two entries are identical to the first two for the host `svr4`: a host-specific route to `slip` through the router `bsdi`, and the loopback route.

The third line is new. It is a direct route (the `G` flag is not set) to a host (the `H` flag is set) and corresponds to our point-to-point link, the SLIP interface. If we compare it to the output from the `ifconfig` command,

```
sun % ifconfig sl0
sl0: flags=1051<UP,POINTOPOINT,RUNNING>
        inet 140.252.1.29 --> 140.252.1.183 netmask ffffff00
```

we see that the destination address in the routing table is the other end of the point-to-point link (the router `netb`) and the gateway address is really the local IP address of the outgoing interface (140.252.1.29). (We said earlier that the gateway address printed by `netstat` for a direct route is the local IP address of the interface to use.)

The default entry is an indirect route (`G` flag) to a network (no `H` flag), as we expect. The gateway address is the address of the router (140.252.1.183, the other end of the SLIP link) and not the local IP address of the SLIP link (140.252.1.29). Again, this is because it is an indirect route, not a direct route.

We should also note that the third and fourth lines output by `netstat` (the ones with an interface of `sl0`) are created by the SLIP software being used when the SLIP line is brought up, and deleted when the SLIP link is brought down.

No Route to Destination

All our examples so far have assumed that the search of the routing table finds a match, even if the match is the default route. What if there is no default route, and a match isn't found for a given destination?

The answer depends on whether the IP datagram being routed was generated on the host or is being forwarded (e.g., we're acting as a router). If the datagram was generated on this host, an error is returned to the application that sent the datagram, either "host unreachable" or "network unreachable." If the datagram was being forwarded, an ICMP host unreachable error is sent back to original sender. We examine this error in the following section.

9.3 ICMP Host and Network Unreachable Errors

The ICMP "host unreachable" error message is sent by a router when it receives an IP datagram that it cannot deliver or forward. (Figure 6.10 shows the format of the ICMP

unreachable messages.) We can see this easily on our network by taking down the dialup SLIP link on the router sun, and trying to send a packet through the SLIP link from any of the other hosts that specify sun as the default router.

> Older implementations of the BSD TCP/IP software generated either a host unreachable, or a network unreachable, depending on whether the destination was on a local subnet or not. 4.4BSD generates only the host unreachable.

Recall from the netstat output for the router sun shown in the previous section that the routing table entries that use the SLIP link are added when the SLIP link is brought up, and deleted when the SLIP link is brought down. This means that when the SLIP link is down, there is no default route on sun. But we don't try to change all the other host's routing tables on our small network, having them also remove their default route. Instead we count on the ICMP host unreachable generated by sun for any packets that it gets that it cannot forward.

We can see this by running ping on svr4, for a host on the other side of the dialup SLIP link (which is down):

```
svr4 % ping gemini
ICMP Host Unreachable from gateway sun (140.252.13.33)
ICMP Host Unreachable from gateway sun (140.252.13.33)
^?                                        type interrupt key to stop
```

Figure 9.2 shows the tcpdump output for this example, run on the host bsdi.

```
1  0.0                  svr4 > gemini: icmp: echo request
2  0.00 (0.00)          sun > svr4: icmp: host gemini unreachable

3  0.99 (0.99)          svr4 > gemini: icmp: echo request
4  0.99 (0.00)          sun > svr4: icmp: host gemini unreachable
```

Figure 9.2 ICMP host unreachable in response to ping.

When the router sun finds no route to the host gemini, it responds to the echo request with a host unreachable.

If we bring the SLIP link to the Internet up, and try to ping an IP address that is not connected to the Internet, we expect an error. What is interesting is to see how far the packet gets into the Internet, before the error is returned:

```
sun % ping 192.82.148.1              this IP address is not connected to the Internet
PING 192.82.148.1: 56 data bytes
ICMP Host Unreachable from gateway enss142.UT.westnet.net (192.31.39.21)
 for icmp from sun (140.252.1.29) to 192.82.148.1
```

Looking at Figure 8.5 (p. 103) we see that the packet made it through six routers before detecting that the IP address was invalid. Only when it got to the border of the NSFNET backbone was the error detected. This implies that the six routers that forwarded the packet were doing so because of default entries, and only when it reached the NSFNET backbone did a router have complete knowledge of every network connected to the Internet. This illustrates that many routers can operate with just partial knowledge of the big picture.

[Ford, Rekhter, and Braun 1993] define a *top-level routing domain* as one that maintains routing information to most Internet sites and does not use default routes. They note that five of these top-level routing domains exist on the Internet: the NFSNET backbone, the Commercial Internet Exchange (CIX), the NASA Science Internet (NSI), SprintLink, and the European IP Backbone (EBONE).

9.4 To Forward or Not to Forward

We've mentioned a few times that hosts are not supposed to forward IP datagrams unless they have been specifically configured as a router. How is this configuration done?

Most Berkeley-derived implementations have a kernel variable named `ipforwarding`, or some similar name. (See Appendix E.) Some systems (BSD/386 and SVR4, for example) only forward datagrams if this variable is nonzero. SunOS 4.1.x allows three values for the variable: –1 means never forward and never change the value of the variable, 0 means don't forward by default but set this variable to 1 when two or more interfaces are up, and 1 means always forward. Solaris 2.x changes the three values to be 0 (never forward), 1 (always forward), and 2 (only forward when two or more interfaces are up).

Older 4.2BSD hosts forwarded datagrams by default, which caused lots of problems for systems configured improperly. That's why this kernel option must always default to "never forward" unless the system administrator specifically enables forwarding.

9.5 ICMP Redirect Errors

The ICMP redirect error is sent by a router to the sender of an IP datagram when the datagram should have been sent to a different router. The concept is simple, as we show in the three steps in Figure 9.3. The only time we'll see an ICMP redirect is when the host has a choice of routers to send the packet to. (Recall the earlier example of this we saw in Figure 7.6, p. 94.)

1. We assume that the host sends an IP datagram to R1. This routing decision is often made because R1 is the default router for the host.
2. R1 receives the datagram and performs a lookup in its routing table and determines that R2 is the correct next-hop router to send the datagram to. When it sends the datagram to R2, R1 detects that it is sending it out the same interface on which the datagram arrived (the LAN to which the host and the two routers are attached). This is the clue to a router that a redirect can be sent to the original sender.
3. R1 sends an ICMP redirect to the host, telling it to send future datagrams to that destination to R2, instead of R1.

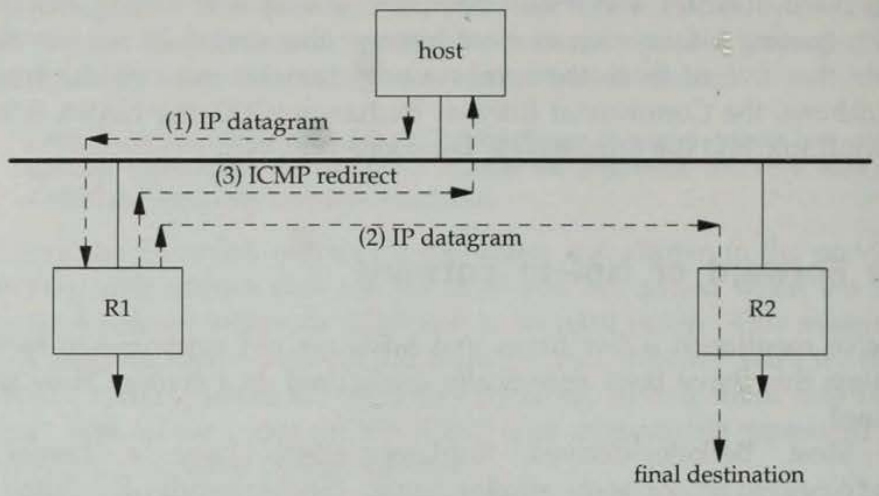


Figure 9.3 Example of an ICMP redirect.

A common use for redirects is to let a host with minimal routing knowledge build up a better routing table over time. The host can start with only a default route (either R1 or R2 from our example in Figure 9.3) and anytime this default turns out to be wrong, it'll be informed by that default router with a redirect, allowing the host to update its routing table accordingly. ICMP redirects allow TCP/IP hosts to be dumb when it comes to routing, with all the intelligence in the routers. Obviously R1 and R2 in our example have to know more about the topology of the attached networks, but all the hosts attached to the LAN can start with a default route and learn more as they receive redirects.

An Example

We can see ICMP redirects in action on our network (inside front cover). Although we show only three hosts (`aix`, `solaris`, and `gemini`) and two routers (`gateway` and `netb`) on the top network, there are more than 150 hosts and 10 other routers on this network. Most of the hosts specify `gateway` as the default router, since it provides access to the Internet.

How is the author's subnet (the bottom four hosts in the figure) accessed from the hosts on the 140.252.1 subnet? First recall that if only a single host is at the end of the SLIP link, proxy ARP is used (Section 4.6). This means nothing special is required for hosts on the top network (140.252.1) to access the host `sun` (140.252.1.29). The proxy ARP software in `netb` handles this.

When a network is at the other end of the SLIP link, however, routing becomes involved. One solution is for every host and router to know that the router `netb` is the gateway for the network 140.252.13. This could be done by either a static route in each host's routing table, or by running a routing daemon in each host. A simpler way (and the method actually used) is to utilize ICMP redirects.

Let's run the `ping` program from the host `solaris` on the top network to the host `bsdi` (140.252.13.35) on the bottom network. Since the subnet IDs are different, proxy ARP can't be used. Assuming a static route has not been installed, the first packet sent will use the default route to the router `gateway`. Here is the routing table before we run `ping`:

```
solaris % netstat -rn
Routing Table:
  Destination       Gateway       Flags  Ref   Use   Interface
---------------- ---------------- ----- ----- ------ ---------
127.0.0.1        127.0.0.1         UH        0    848  lo0
140.252.1.0      140.252.1.32      U         3  15042  le0
224.0.0.0        140.252.1.32      U         3      0  le0
default          140.252.1.4       UG        0   5747
```

(The entry for 224.0.0.0 is for IP multicasting. We describe it in Chapter 12.) If we specify the `-v` option to `ping`, we'll see any ICMP messages received by the host. We need to specify this to see the redirect message that's sent.

```
solaris % ping -sv bsdi
PING bsdi: 56 data bytes
ICMP Host redirect from gateway gateway (140.252.1.4)
 to netb (140.252.1.183) for bsdi (140.252.13.35)
64 bytes from bsdi (140.252.13.35): icmp_seq=0. time=383. ms
64 bytes from bsdi (140.252.13.35): icmp_seq=1. time=364. ms
64 bytes from bsdi (140.252.13.35): icmp_seq=2. time=353. ms
^?                                    type interrupt key to stop
----bsdi PING Statistics----
4 packets transmitted, 3 packets received, 25% packet loss
round-trip (ms)  min/avg/max = 353/366/383
```

Before we receive the first `ping` response, the host receives an ICMP redirect from the default router `gateway`. If we then look at the routing table, we'll see that the new route to the host `bsdi` has been inserted. (This new entry is shown in a bolder font.)

```
solaris % netstat -rn
Routing Table:
  Destination       Gateway       Flags  Ref   Use   Interface
---------------- ---------------- ----- ----- ------ ---------
127.0.0.1        127.0.0.1         UH        0    848  lo0
140.252.13.35    140.252.1.183     UGHD      0      2
140.252.1.0      140.252.1.32      U         3  15045  le0
224.0.0.0        140.252.1.32      U         3      0  le0
default          140.252.1.4       UG        0   5749
```

This is the first time we've seen the `D` flag, which means the route was installed by an ICMP redirect. The G flag means it's an indirect route to a gateway (`netb`), and the `H` flag means it's a host route (as we expect), not a network route.

Since this is a host route, added by a host redirect, it handles only the host `bsdi`. If we then access the host `svr4`, another redirect is generated, creating another host route. Similarly, accessing the host `slip` creates another host route. The point here is that each redirect is for a single host, causing a host route to be added. All three hosts on the author's subnet (`bsdi`, `svr4`, and `slip`) could also be handled by a single network

route pointing to the router sun. But ICMP redirects create host routes, not network routes, because the router generating the redirect in this example (gateway) has no knowledge of the subnet structure on the 140.252.13 network.

More Details

Figure 9.4 shows the format of the ICMP redirect message.

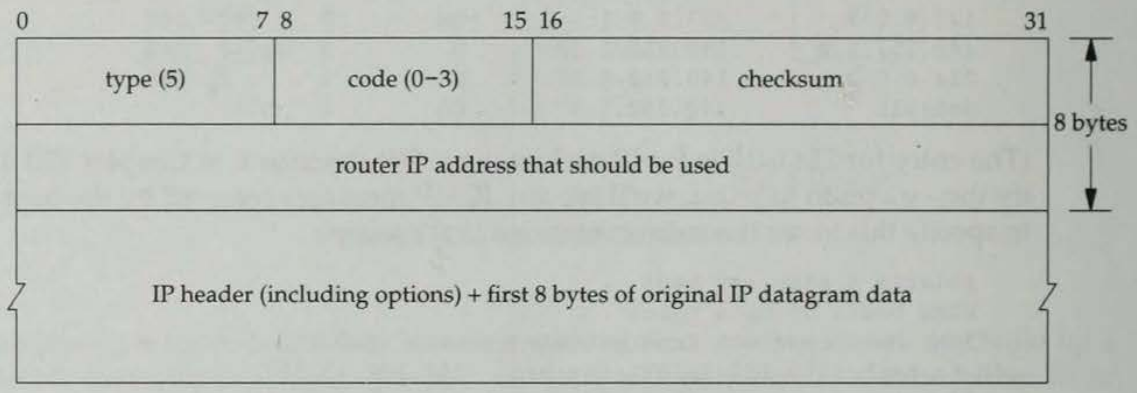


Figure 9.4 ICMP redirect message.

There are four different redirect messages, with different *code* values, as shown in Figure 9.5.

code	Description
0	redirect for network
1	redirect for host
2	redirect for type-of-service and network
3	redirect for type-of-service and host

Figure 9.5 Different *code* values for ICMP redirect.

There are three IP addresses that the receiver of an ICMP redirect must look at: (1) the IP address that caused the redirect (which is in the IP header returned as the data portion of the ICMP redirect), (2) the IP address of the router that sent the redirect (which is the source IP address of the IP datagram containing the redirect), and (3) the IP address of the router that should be used (which is in bytes 4–7 of the ICMP message).

There are numerous rules about ICMP redirects. First, redirects are generated only by routers, not by hosts. Also, redirects are intended to be used by hosts, not routers. It is assumed that routers participate in a routing protocol with other routers, and the routing protocol should obviate the need for redirects. (This means that in Figure 9.1 the routing table should be updated by either a routing daemon or redirects, but not by both.)

4.4BSD, when acting as a router, performs the following checks, all of which must be true before an ICMP redirect is generated.

1. The outgoing interface must equal the incoming interface.
2. The route being used for the outgoing datagram must not have been created or modified by an ICMP redirect, and must not be the router's default route.
3. The datagram must not be source routed.
4. The kernel must be configured to send redirects.

> The kernel variable is named `ip_sendredirects`, or something similar. (See Appendix E.) Most current systems (4.4BSD, SunOS 4.1.x, Solaris 2.x, and AIX 3.2.2, for example) enable this variable by default. Other systems such as SVR4 disable it by default.

Additionally, a 4.4BSD host that receives an ICMP redirect performs some checks before modifying its routing table. These are to prevent a misbehaving router or host, or a malicious user, from incorrectly modifying a system's routing table.

1. The new router must be on a directly connected network.
2. The redirect must be from the current router for that destination.
3. The redirect cannot tell the host to use itself as the router.
4. The route that's being modified must be an indirect route.

Our final point about redirects is that routers should send only host redirects (*code*s 1 or 3 from Figure 9.5) and not network redirects. Subnetting makes it hard to specify exactly when a network redirect can be sent instead of a host redirect. Some hosts treat a received network redirect as a host redirect, in case a router sends the wrong type.

9.6 ICMP Router Discovery Messages

We mentioned earlier in this chapter that one way to initialize a routing table is with static routes specified in configuration files. This is often used to set a default entry. A newer way is to use the ICMP router advertisement and solicitation messages.

The general concept is that after bootstrapping, a host broadcasts or multicasts a router solicitation message. One or more routers respond with a router advertisement message. Additionally, the routers periodically broadcast or multicast their router advertisements, allowing any hosts that are listening to update their routing table accordingly.

RFC 1256 [Deering 1991] specifies the format of these two ICMP messages. Figure 9.6 shows the format of the ICMP router solicitation message. Figure 9.7 shows the format of the ICMP router advertisement message sent by routers.

Multiple addresses can be advertised by a router in a single message. *Number of addresses* is the number. *Address entry size* is the number of 32-bit words for each router address, and is always 2. *Lifetime* is the number of seconds that the advertised addresses can be considered valid.

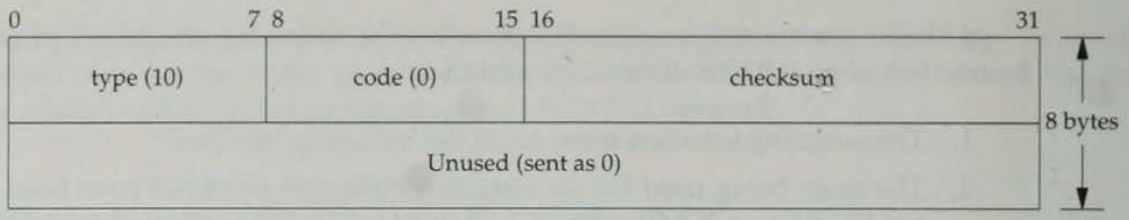


Figure 9.6 Format of ICMP router solicitation message.

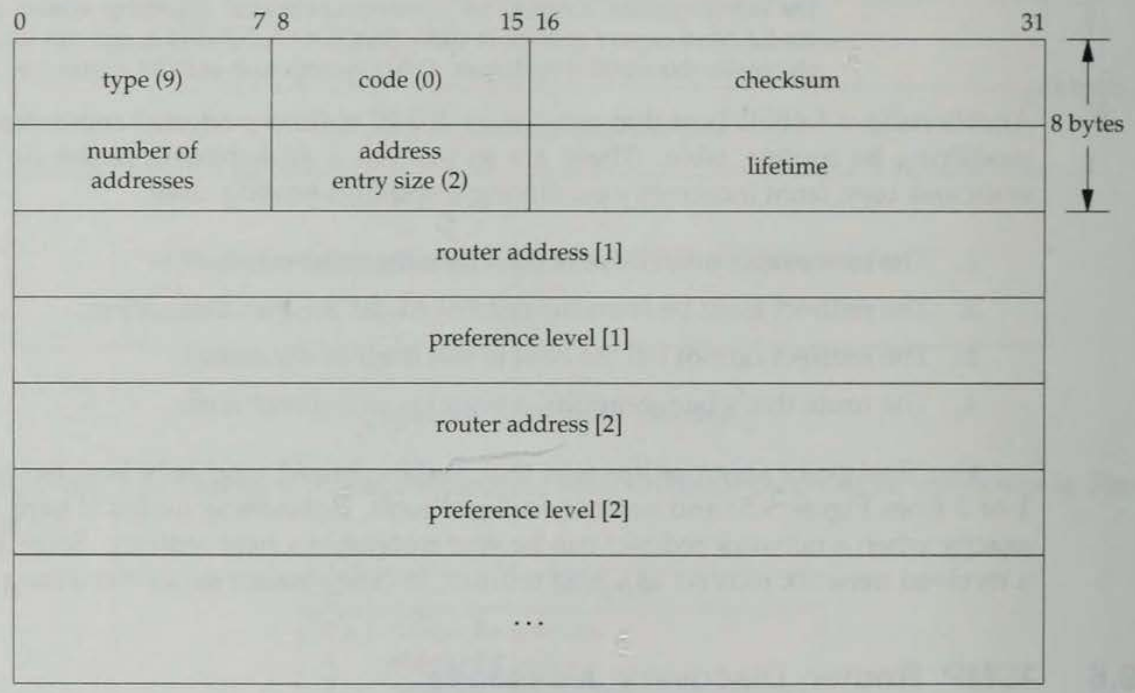


Figure 9.7 Format of ICMP router advertisement message.

One or more pairs of an IP address and a preference then follow. The IP address must be one of the sending router's IP addresses. The *preference level* is a signed 32-bit integer indicating the preference of this address as a default router address, relative to other router addresses on the same subnet. Larger values imply more preferable addresses. The preference level `0x80000000` means the corresponding address, although advertised, is not to be used by the receiver as a default router address. The default value of the preference is normally 0.

Router Operation

When a router starts up it transmits periodic advertisements on all interfaces capable of broadcasting or multicasting. These advertisements are not exactly periodic, but are

randomized, to reduce the probability of synchronization with other routers on the same subnet. The normal time interval between advertisements is between 450 and 600 seconds. The default lifetime for a given advertisement is 30 minutes.

Another use of the lifetime field occurs when an interface on a router is disabled. In that case the router can transmit a final advertisement on the interface with the lifetime set to 0.

In addition to the periodic, unsolicited advertisements, a router also listens for solicitations from hosts. It responds to these solicitations with a router advertisement.

If there are multiple routers on a given subnet, it is up to the system administrator to configure the preference level for each router as appropriate. For example, the primary default router would have a higher preference than a backup.

Host Operation

Upon bootstrap a host normally transmits three router solicitations, 3 seconds apart. As soon as a valid advertisement is received, the solicitations stop.

A host also listens for advertisements from adjacent routers. These advertisements can cause the host's default router to change. Also, if an advertisement is not received for the current default, that default can time out.

As long as the normal default router stays up, that router will send advertisements every 10 minutes, with a lifetime of 30 minutes. This means the host's default entry won't time out, even if one or two advertisements are lost.

Implementation

The router discovery messages are normally generated by and processed by a user process (a daemon). This adds yet another program updating the routing table in Figure 9.1, although it would only add or delete a default entry. The daemon would have to be configured to act as a router or a host.

> These two ICMP messages are new and not supported by all systems. Solaris 2.x is the only system in our network that supports these messages (the `in.rdisc` daemon). Although the RFC recommends using IP multicasting whenever possible, router discovery can work using broadcast messages also.

9.7 Summary

The operation of IP routing is fundamental to a system running TCP/IP, be it a host or router. The routing table entries are simple: up to 5 flag bits, a destination IP address (host, network, or default), a next-hop router IP address (for an indirect route) or a local interface IP address (for a direct route), and a pointer to a local interface to use. Host entries have priority over network entries, which have priority over default entries.

A search of this routing table is made for every IP datagram that the system generates or forwards, and can be updated by either a routing daemon or ICMP redirects. By default a system should never forward a datagram unless it has specifically been

configured to do so. Static routes can be entered using the `route` command, and the newer ICMP router discovery messages can be used to initialize and dynamically update default entries. Hosts can start with a simple routing table that is updated dynamically by ICMP redirects from its default router.

Our discussion in this chapter has focused on how a single system uses its routing table. In the next chapter we examine how routers exchange routing information with each other.

Exercises

9.1 Why do you think both types of ICMP redirects—network and host—exist?

9.2 In the routing table for `svr4` shown at the beginning of Section 9.2, is a specific route to the host `slip` (140.252.13.65) necessary? What would change if this entry were removed from the routing table?

9.3 Consider a cable with both 4.2BSD hosts and 4.3BSD hosts. Assume the network ID is 140.1. The 4.2BSD hosts only recognize a host ID of all zero bits as the broadcast address (140.1.0.0), while the 4.3BSD hosts normally send a broadcast using a host ID of all one bits (140.1.255.255). Also, the 4.2BSD hosts by default will try to forward incoming datagrams, even if they have only a single interface.

Describe the events that happen when the 4.2BSD hosts receives an IP datagram with the destination address of 140.1.255.255.

9.4 Continue the previous exercise, assuming someone corrects this problem by adding an entry to the ARP cache on one system on the 140.1 subnet (using the `arp` command) saying that the IP address 140.1.255.255 has a corresponding Ethernet address of all one bits (the Ethernet broadcast). Describe what happens now.

9.5 Examine your system's routing table and describe each entry.